HOCHSCHULE MITTWEIDA
UNIVERSITY OF APPLIED SCIENCES

BACHELORARBEIT

Herr
Lucas Gering

Nicht-räumliche Parallelisierung von 3D-Visualisierung – Pro und Kontra

2016

angewandte Computer- und Biowissenschaften

BACHELORARBEIT

Nicht-räumliche Parallelisierung von 3D-Visualisierung – Pro und Kontra

Autor:
Herr Lucas Gering

Studiengang:
Medieninformatik u. i. E.

Seminargruppe:
MI11w1 - B

Erstprüfer:
Prof. Dr. rer. nat. habil. Thomas Haenselmann

Zweitprüfer:
Prof. Dr. rer. Nat. Christian Hummert

Einreichung:
Mittweida, 11.04.2016

angewandte Computer- und Biowissenschaften

BACHELOR THESIS

Not spatial parallelization of 3D visualization – pros and cons

author:
Mr. Lucas Gering

course of studies:
Medieninformatik u. i. E.

seminar group:
MI11w1 - B

first examiner:
Prof. Dr. rer. nat. habil. Thomas Haenselmann

second examiner:
Prof. Dr. rer. nat. Christian Hummert

submission:
Mittweida, 11.04.2016

Bibliografische Angaben

Gering, Lucas:

Nicht-räumliche Parallelisierung von 3D-Visualisierung – Pro und Kontra

57 Seiten, Hochschule Mittweida, University of Applied Sciences,
Angewandte Computer-und Biowissenschaften, Bachelorarbeit, 2016

Inhaltsverzeichnis

Inhaltsverzeichnis V

Abkürzungsverzeichnis VII

Abbildungsverzeichnis VIII

1 Motivation dieser Arbeit 1

2 Bildsynthese – ein Überblick 2

2.1 Was versteht man unter Bildsynthese 2

2.2 Polygonen-Renderer 3

2.2.2 Lichtberechnungsansätze 8

2.3 Raytracing 11

2.4 Potentielle Einsatzszenarien für Rendersysteme 13

3 Welche Herausforderungen stellt Paralleles Rendern dar? 14

3.1 Potentielle Hardware Szenarien für Paralleles Rendern 14

3.1.1 Multi-GPU Systeme 15

3.1.2 Netzwerk Rendersysteme 16

4 Verteilungsansätze 17

4.1 Kohärenz als Faktor der Parallelisierung 17

4.2 Die Funktionsweise der Objekt-Parallelisierung 18

4.3 Die Funktionsweise der Bild-Parallelisierung 19

4.4 Die Funktionsweise der Frame-Verteilung 20

4.5 Vergleich dieser Ansätze 22

5 Parallelisierungsverfahren und ihre Klassifizierung 24

5.1 Die Gruppe der Sort-First-Algorithmen

und ihre allgemeine Arbeitsweise 26

5.2 Die Gruppe der Sort-Middle-Algorithmen

und ihre allgemeine Arbeitsweise 28

5.3 Die Gruppe der Sort-Last-Algorithmen

und ihre allgemeine Arbeitsweise 29

5.4 Der Hybrid-Sort-First-Sort-Last-Algorithmus 31

5.5 Zusatz Verteilung in einem Hybrid Sort-First-Sort-Last-Verfahren 35

6 Umsetzung eines Hybrid Verteilungsverfahren an einem Beispiel 36

6.1 Ausgangssituation.....36

6.2 Entwicklung des Verteilungsverfahrens und Implementierung in das bestehende Programm.....37

6.3 Fazit der Implementierung.....42

7 Fazit.....44

Literaturverzeichnis XI

Anlagen.....XIV

Eigenständigkeitserklärung XV

Abkürzungsverzeichnis

2D	zweidimensional
3D	dreidimensional
CPU	Central Processing Unit
GPU	Graphics Processing Unit
SLI	Scalable-Link-Interface
PCI	Peripheral Component Interconnect
SFR	Split-Frame-Rendering
AFR	Alternate-Frame-Rendering

Abbildungsverzeichnis

Abbildung 1: Schematische Darstellung einer Renderpipeline 3
Abbildung 2: Lichtberechnung im Vertex Shader (linke Kapsel) umd im Pixel Shader (rechte Kapsel) mit Wireframe der Kapselgeometrie 9
Abbildung 3: die zwei Framebuffer einer Defferd-Renderingpipeline und die resultierende Beleuchtung 10
Abbildung 4: Projektion eines Polygons auf die Betrachterebene 11
Abbildung 5: die Beleuchtung eines Polygons durch eine Lichtquelle 12
Abbildung 6: Schematische Darstellung eines Objektverteilungsverfahrens auf zwei Grafikkarten 19
Abbildung 7: Schematische Darstellung eines Split-Frame-Renderings mit zwei Grafikkarten 20
Abbildung 8: Schematische Darstellung eines Alternate-Frame-Rendering Prozesses auf zwei Grafikkarten 21
Abbildung 9: Teilabschnitt einer Renderpipeline mit einer Sort-First-Verteilung vor dem Geometrieschritt 24
Abbildung 10: Teilabschnitt einer Renderpipeline mit einer Sort-Middle-Verteilung zwischen dem Geometrie- und Rasterschritt 25
Abbildung 11: Teilabschnitt einer Renderpipeline mit einer Sort-Last-Verteilung nach dem Rasterschritt 25
Abbildung 12: Schematische Darstellung der drei Bearbeitungsphasen eines Hybrid Verteilungsverfahrens 32
Abbildung 13: drei Momentaufnahmen auf dem Verteilungsvorgangs eines Hybrid-Verteilungsverfahrens 33
Abbildung 14: Quelltext des Object Structs 37
Abbildung 15: ThreadData Struct für den Datenaustausch zwischen der main Funktion und dem Thread 38
Abbildung 16: PreTransformThreadDate für den Datenaustausch zwischen main Funktion und den Pre-Transformation Threads 40
Abbildung 17: Funktion zur Erstellung der Teilaufgaben HybridSubTaskMaker 41

1 Motivation dieser Arbeit

In den vergangenen Jahren nahm die Bedeutung von computergenerierten Bildern stark zu. Viele Anwendungen im Unterhaltungsbereich, auch in der Visualisierung, nutzen sowohl vorberechnete als auch interaktive dreidimensionale Darstellungen. Der Anspruch, diese Inhalte möglichst komplex und realitätsnah darzustellen, führt zwangsläufig zu einem gesteigerten Bedarf an Ressourcen. Dabei erreichen einzelne Berechnungseinheiten schnell ihr Leistungsgrenzen. Es liegt daher nahe, mehrere Recheneinheiten gleichzeitig für die Berechnung komplexer Darstellungen zu nutzen. Dabei ergibt sich jedoch eine zusätzliche Problemstellung, die Verteilung der Aufgaben auf die verfügbaren Recheneinheiten[18]. In der Nichtechtzeitgrafik hat sich vor allem ein Verfahren durchgesetzt, das einzelnen Frames auf einzelne Recheneinheiten verteilt. Die Recheneinheiten errechnen getrennt voneinander die einzelnen Bilder und setzten sie anschließend zu einem Video zusammen. Dieses Verfahren erscheint bei genauerer Betrachtung jedoch sehr ineffizient, da Frames unterschiedlich komplex seinen können und keine Möglichkeit besteht, aus voran gegangenen Frames Daten zu übernehmen. Dies könnte einer optimalen Ausnutzung der vorhandenen Ressourcen entgegenstehen.

Die nachfolgende Arbeit befasst sich mit alternativen Möglichkeiten einen Rendervorgang auf mehrere Recheneinheiten zu verteilen. Im Fokus stehen dabei Verfahren, die aktiv den Inhalt der Szene analysieren und in die Aufgabenverteilung einzubeziehen. Die Arbeit stellt eine Auswahl verschiedener Verfahren vor und vergleicht diese hinsichtlich ihrer Eigenschaften. Exemplarisch wird ein Hybrid-Sort-First-Sort-Last Verfahren in C++ realisiert. Abschließend wird die Frage gestellt, welche Verteilungsverfahren in bestimmten Anwendungsgebieten den gängigen Verfahren überlegen sind und ob nicht räumliche Verteilungen einen Mehrwert darstellen.

2 Bildsynthese – ein Überblick

Der Ansatzpunkt für die Verteilung einer grafischen Berechnung auf mehrere Recheneinheiten liegt in der Bildsynthese selbst. Der nachfolgende Abschnitt thematisiert die unterschiedlichen Möglichkeiten, ein computergeneriertes Bild zu erzeugen. Besonderer Schwerpunkt liegt dabei auf der Funktionsweise der so genannten Polygonen-Renderer, da diese die gebräuchlichste Form eines Renderers darstellen.

2.1 Was versteht man unter Bildsynthese

Bildsynthese beschreibt in der Computergrafik das Erzeugen eines Bildes aus Rohdaten der datentechnischen Repräsentation einer virtuellen Umgebung. Diese virtuelle Umgebung mit Objekten, Lichtquellen, Materialeigenschaften und Betrachter-Informationen wird auch Szene genannt. Ein Computerprogramm, welches diese Darstellung aus einem Datensatz errechnet und sichtbar darstellt, wird als Renderer bezeichnet.

Renderer lassen sich dabei in verschiedene Kategorien klassifizieren. Ein Kriterium kann dabei die Anzahl der Dimensionen sein, in denen die ursprüngliche Szene vorliegt. Derzeit beschränken sich Renderer auf den zweidimensionalen und dreidimensionalen Raum. Dabei bleibt das erzeugte Bild jedoch immer zweidimensional und vermittelt lediglich den Eindruck einer räumlichen Darstellung.

Eine andere Unterteilung stellt die beabsichtigte Berechnungsdauer pro Einzelbild dar. Daraus ergeben sich zwei mögliche Kategorien. Ein Echtzeitrenderer legt seinen Fokus auf die zeitnahe Berechnung und Anzeige einzelner Frames, um eine flüssige Darstellung zu gewährleisten. In der Praxis müssen dafür mindestens 24 Bilder pro Sekunde errechnet werden, um den Eindruck einer Bewegung zu vermitteln. Die Bildqualität richtet sich dabei nach der Leistungsfähigkeit der Recheneinheiten und ist von untergeordneter Priorität. Ein Nichtechtzeitrenderer hingegen legt den Schwerpunkt auf die hohe Qualität der zu berechnenden Bilder. Häufig ist der Anspruch an einen solchen Renderer, ein möglichst realitätsnahes Bild zu erzeugen und somit zum Beispiel Licht physikalisch korrekt zu berechnen. Die Bearbeitungszeit ist dabei von untergeordneter Bedeutung, da die gerenderten Bilder meist zwischengespeichert und erst zu einem späteren Zeitpunkt wiedergegeben werden.

Der wesentliche Unterscheidungspunkt liegt jedoch im Ansatz für die Lichtberechnung innerhalb der Szene. Diese Lichtberechnung stellt in den meisten Szenarien den rechentechnisch aufwändigsten Teilschritt dar und ist somit auch ausschlaggebend für

die spätere Qualität und Berechnungsdauer. Die drei gängigsten Berechnungsverfahren für dreidimensionale Szenen sind Polygonen-Renderer, Raytracer und Radiosity-Renderer. Renderer für den Einsatz in zweidimensionalen Szenarien sind kein Bestandteil dieser Arbeit und werden daher nicht explizit behandelt. Es ist jedoch möglich, die vorgestellten Techniken in abgewandelter Form auch auf 2D-Renderer anzuwenden.

2.2 Polygonen-Renderer

Polygonen-Renderer stellen die gebräuchlichste Gruppe der 3D-Renderer dar und erzeugen unter Echtzeitbedingungen eine hohe, wenn auch nicht physikalisch korrekt errechnete Bildqualität. Daher finden sie ihr Haupteinsatzgebiet insbesondere im Bereich der Echtzeitgrafikberechnung sowie bei interaktiven Anwendungen.

Die Funktionsweise eines Polygonen-Renderers orientiert sich am Prinzip der Renderpipeline. Eine Renderpipeline beschreibt den Prozess, beim dem ausgehend von Szenendaten in aufeinander aufbauenden Teilschritten ein fertiges Bild erzeugt wird. Abbildung 1 zeigt den vereinfachten Aufbau einer klassischen Renderpipeline[13].

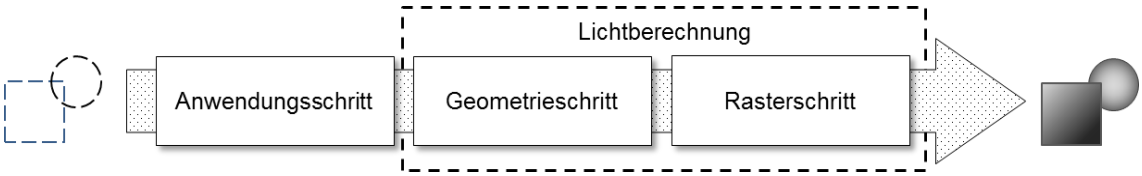

Abbildung 1: Schematische Darstellung einer Renderpipeline

Im ersten Teilschritt der Renderpipeline, dem Anwendungsschritt, wird die Szene für den Rendervorgang vorbereitet. Seine Hauptaufgabe liegt in der Umwandlung aller Daten der 3D Szene in Positionsangaben für Vertexpunkte und zusätzliche Hilfsobjekte. Als Vertexpunkt (engl. für Eckpunkt oder Knoten) bezeichnet man die Sonderform eines geometrischen Punktes, der einen Eckpunkt einer Fläche oder Schnittpunkt mehrerer geometrischer Formen repräsentiert. Drei oder mehr Vertexpunkte spannen mit Strecken, sogenannten Edges, verbunden eine Fläche im Raum auf. Diese Flächen nennt man auch Polygon. Eine zusammenhängende Gruppe mehrerer Polygonen bildet ein Objekt.

Im darauf folgenden Geometrieschritt werden die Polygone jedes Objektes im Raum betrachtet. Ziel ist es, dabei jede Fläche im Sichtbereich auf die zweidimensionale Bildebene des Betrachters zu projizieren. Die Szene liegt anschließend als Pixeldaten vor. Abhängig von der gewählten Rendertechnik kann bereits während des Geometrieschritts eine erste Lichtberechnung der Szene erfolgen. Diese kann jedoch auch mit erheblich höherer Qualität erst während des Rasterschritts durchgeführt werden.

Im Rasterschritt, dem letzten Schritt einer Renderpipeline, wird die Szene auf Pixelebene betrachtet. Dabei werden die unterschiedlichen Pixeldaten auf einen einzelnen Farbwert reduziert und zu einem zusammenhängenden Bild verarbeitet. Während dieses Teilschrittes werden zusätzlich alle Postprocessing Effekte auf das Bild angewendet. Diese Effekte sind optische Verfahren, bei denen der entsprechende Frame nur mit Hilfe der Pixeldaten bearbeitet wird. Typische Postprocessing Effekte sind Farbkorrekturen, Überstrahleffekte (Bloom), Bewegungsunschärfe und das Simulieren einer virtuellen Kameralinse mit zum Beispiel Lens Flares und Linsenkrümmung.

Die Anzahl und Aufgaben der Teilschritte können sich je nach Rendertechnik unterscheiden. Eine Unterteilung ist, ob es sich um eine Forward- oder Deferred-Renderpipeline handelt. Eine Forward-Renderpipeline führt die Lichtberechnung während des Geometrie- und Rasterschritts durch (siehe Abbildung 1) und bearbeitet dabei jedes Objekt innerhalb des Sichtfeldes des Betrachters. Im Gegensatz dazu findet bei einer Deferred-Renderpipeline die Lichtberechnung erst nach der Projektion der Szene auf die Betrachterebene statt. Die Lichtberechnung erfolgt somit nur für jeden sichtbaren Pixel des zu rendernden Bildes. Dazu wird bei einer Deferred-Renderpipeline nachfolgend zum Rasterschritt eine weitere Bearbeitungsphase angehängt[14].

Nachfolgend werden die einzelnen Schritte der Renderpipeline und deren Funktionsweise genauer erläutert. Es wird betrachtet, wie die Forward- und Deferred-Renderpipeline im Detail funktionieren und welche Unterschiede zwischen diesen Rendertechniken bestehen.

Der Anwendungsschritt

Nicht alle Objekte einer Szene liegen direkt als Vertexpunkte vor. Viele Vertexpunkte müssen zunächst aus Rohdaten errechnet werden, da im Geometrieschritt lediglich Vertexpunkte und Hilfsobjekte verarbeitet werden können. Diese Vorgehensweise bietet die Möglichkeit, zum Beispiel bei Animationen Speicherplatz zu sparen und gleichzeitig bis unmittelbar vor dem Rendervorgang Eigenschaften der Objekte zu

modifizieren. Die Hauptaufgabe des Anwendungsschritts liegt daher in allen vorbereitenden Berechnungen vor dem Renderprozess. Gleichwohl ist der Anwendungsschritt Bestandteil des Renderprozesses und der Renderpipeline.

In vielen 3D Programmen werden Bewegungen über die Interpolation zwischen sogenannten Keyframes erzeugt. Diese Keyframes stellen meistens Schlüsselpositionen einer Bewegung dar. Die Bewegung des Objektes in der Zeit zwischen zwei Keyframes wird entsprechend einer Interpolationskurve dargestellt. Die Position des Objektes wird im Anwendungsschritt für jeden einzelnen Frame berechnet und von der Bewegung des gesamten Objekts in eine Positionsänderung für jeden einzelnen Vertexpunkt dieses Objektes umgewandelt. Die Bewegung wird somit zu einer Momentaufnahme für jeden einzelnen Frame.

Weiterhin werden während des Anwendungsschrittes alle Modifikatoren und prozeduralen Inhalte errechnet. Prozedurale Inhalte sind Objekte oder Texturen, die erst während des Rendervorganges aus einem Datensatz errechnet werden. Sie sparen so Speicherplatz und können gleichzeitig für die unterschiedlichen Anforderungen jedes einzelnen Anwenders angepasst werden. Modifikatoren hingegen verändern das Aussehen eines vordefinierten Objektes auf Basis verschiedener Optionen. So können sie zum Beispiel auf ein animiertes Objekt angewendet werden und dieses abseits der eigentlichen Animation beeinflussen. Ein typischer Modifikator ist der Spiegel-Modifikator. Dieser kopiert das vorgegebene Objekt und spiegelt es um eine zuvor definierte Achse. Damit kann ein symmetrisches Objekt platzsparender gespeichert werden. Je nach 3D Programm werden viele Bearbeitungsschritte, die aus einem Grundkörper das finale 3D Modell erzeugen, als Modifikatoren gespeichert. So lassen die übereinander „gestapelten“ Modifikatoren auch Veränderungen an den einzelnen Bearbeitungsschritten der Geometrie zu. Für den Renderer werden diese Änderungen zu Positionsdaten der Vertexpunkte umgewandelt. Diese können anschließend weiterbearbeitet und dargestellt werden.

Auch Simulationen wie Stoff, Gravitation, Zerstörung und Wasser müssen für jeden Frame berechnet werden und in Positionsänderungen umgewandelt werden. In vielen Programmen werden diese Simulationen in Keyframes hinterlegt und lassen sich auf ähnlichem Weg wie Animationen verarbeiten. Am Ende des Anwendungsschrittes steht eine 3D Szene, die ausschließlich aus der Szenengeometrie als Punktwolke und Hilfsobjekten wie Lichtquellen besteht. Alle Bearbeitungsschritte während dem Erstellen der 3D Szene wurden auf die einzelnen Vertexpunkte angewendet und liegen nun entweder als statisches Objekt oder als Vertex-Animation vor. Für die Szene kann nun der eigentliche Bildsynthesevorgang durchgeführt werden.

Der Geometrieschritt

Im Geometrieschritt findet die Berechnung der Szene auf der Polygonen- und der Vertexpunktebene statt. Dazu wird in der Theorie jedes Polygon und jeder Vertexpunkt der Szene in fünf Teilschritten weiterverarbeitet. In der Praxis wird jedoch eine Reihe von Algorithmen eingesetzt, um die Anzahl der zu bearbeiteten Objekte zu reduzieren und so die Berechnungen dieses Frames zu beschleunigen[13].

Im ersten Teilschritt findet die Umwandlung aller lokalen Koordinaten innerhalb eines Objektes in das globale Koordinatensystem der Szene statt. Die Vertexpunkte eines Objektes werden in den meisten Fällen in einem Koordinatensystem mit dem Ursprung im Objektankerpunkt angegeben. Die Umwandlung erfolgt, indem aus den Koordinaten der Punkte zunächst eine Positionsmatrize erzeugt wird. Diese wird anschließend mit den drei Rotationsmatrizen der drei Achsen der Szene multipliziert. Anschließend wird eine Translationsmatrize angewendet, um die endgültige globale Position des Vertexpunktes zu erhalten. Um Rechenaufwand während dieser Berechnung zu sparen, werden die drei Rotationsmatrizen und die Translationsmatrize bereits vor dem Rendervorgang miteinander multipliziert. Die erhaltene Umrechnungsmatrize muss anschließend nur noch gespeichert werden und reduziert so die Anzahl der Multiplikationsschritte auf einen einzigen.

In vielen Programmen wird die Szene noch zusätzlich umgebaut, um nachfolgende Berechnungen zu vereinfachen. Dazu wird die Szene so verschoben, dass der Betrachter (die Szenekamera) im Koordinatenursprung liegt. Die Blickrichtung der Kamera wird dabei als die z-Achse definiert. Dieser Vorgang erleichtert später die Projektion der Szene auf die Betrachterebene und das Objekt Clipping.

Während der zweiten Phase des Geometrieschritts kann je nach Lichtberechnungsverfahren eine erste grobe Beleuchtung der Szene durchgeführt werden.

In der dritten Phase wird die 3D Szene auf die zweidimensionale Projektionsfläche der Kamera übertragen. Dazu werden zunächst zwei Clipping Planes erzeugt. Clipping Planes sind Ebenen im Raum, die die Position markieren, ab der Objekte von dem Renderer dargestellt oder ausgeblendet werden. Diese Clipping Planes spannen einen pyramidenstumpfähnlichen Sichtbereich der Kamera auf. Alle Objekte außerhalb dieses Bereiches werden nicht in die Darstellung einbezogen. Meistens soll die Szene perspektivisch dargestellt werden und es wird die Zentralprojektion verwendet. Neben dem eigentlichen Projektionsbild wird auch ein Tiefenkanal (Z-Buffer) erzeugt. Dieser speichert in einem Graustufenbild die Tiefe jedes Punktes auf der Projektionsfläche.

Im vierten Teilschritt werden zunächst die nicht sichtbaren Rückseiten der Objekte ausgeblendet. Weiterhin müssen alle Polygone, die sich nur zu einem Teil außerhalb des Sichtfeldes befinden, mittels Lineclipping auf die sichtbare Fläche reduziert werden. Dazu wird ein Polygon entlang des sichtbaren Teils in zwei neue Polygone geteilt. Das nicht sichtbare Polygon wird verworfen. Das sichtbare Polygon schließt bündig mit dem Bildrand oder verdeckenden Objekt ab.

Im letzten Teilschritt findet die Window-Viewport-Transformation statt, in der das entstandene Bild an die Seitenverhältnisse und Größe der Ausgabe angepasst wird. Dieser Vorgang wird nicht benötigt, wenn das gerenderte Bild bereits die gleiche Größe besitzt wie die Ausgabe.

Der Rasterschritt

Im Rasterschritt wird jede sichtbare Fläche auf dem Bildschirm in kleinere Einheiten (Fragmente) zerlegt. Jedes Fragment entspricht dabei einem Pixel auf dem Bildschirm. Zunächst wird der Z-Buffer (Tiefenkanal), der während des Geometrieschritts erzeugt wird, auf das Bild angewendet. Mit diesem Graustufenbild wird für jeden Pixel bestimmt, welches Objekt räumlich näher an der Kamera liegt und daher gezeichnet werden muss. Neben den eigentlichen Szenenberechnungen finden auch das Postprocessing und die Antialiasing Berechnungen statt. Postprocessing beschreibt alle optischen Verfahren, mit denen der Frame vor der endgültigen Bildausgabe bearbeitet werden kann. Typische Effekte während des Postprocessings sind Farbkorrekturen, Bewegungsunschärfe, Tiefenschärfe und Ambient Occlusion. Bei Bedarf wird während des Rasterschritts auch die Treppchenbildung mittels Antialiasing unterdrückt. Am Ende des Rasterschritts steht das Bild in der gewünschten Auflösung bereit. Bei der Ausgabe über eine Grafikkarte wird das finale Bild zunächst in einem Zwischenspeicher abgelegt, dem sogenannten Backbuffer. Wenn das Bild bereit für die Ausgabe ist, wird der Backbuffer mit dem aktuell dargestellten Frontbuffer vertauscht. Das gerenderte Bild wird ausgegeben. Dieses Verfahren verhindert bei langwierigeren Postprocessing Arbeiten, dass ein noch nicht vollständig berechnetes Bild bereits dargestellt wird und gleichzeitig noch Teile des vorangegangenen Frames sichtbar sind. Diesen Effekt wird Screen Tearing genannt[15].

2.2.2 Lichtberechnungsansätze

Eine der aufwändigsten und zugleich wichtigsten Aufgaben der Renderer ist das Simulieren des Lichtverhaltens innerhalb der virtuellen Szene. Dieser Vorgang umfasst dabei vor allem den Licht- und Schattenwurf in der Szene. In der Praxis haben sich je nach Einsatzgebiet zwei unterschiedliche Ansätze für die Lichtberechnung innerhalb der Renderpipeline durchgesetzt.

Eine Forward-Renderingpipeline führt die Lichtberechnung während des Geometrie- und Rasterschritts für jeden Vertexpunkt oder Pixel der Szene durch. Im Gegensatz dazu findet die Lichtberechnung bei einer Deferred-Renderingpipeline in einem zusätzlichen Pipelineschritt nach dem Rasterschritt statt. Diese Lichtberechnung wird nur für jeden sichtbaren Pixel des zu rendernden Frames durchgeführt.

Forward Rendering Pipeline

Die Forward-Renderingpipeline orientiert sich von ihrem Aufbau an einer klassischen Renderpipeline, wie sie im vorangegangen Abschnitt beschrieben wurde. Wenn die Lichtberechnung während des Geometrieschritts stattfindet, wird für jedes Polygon die entsprechende Oberflächennormale bestimmt und gespeichert. Anschließend wird das Skalarprodukt zwischen der Oberflächennormale und der Normale der Lichtquelle berechnet. Ein negatives Skalarprodukt zeigt, dass dieses Polygon der Lichtquelle abgewandt ist und nicht von ihr beleuchtet wird. Ein positives Skalarprodukt hingegen bedeutet, dass das Polygon der Lichtquelle zugewandt ist und beleuchtet wird. Bei einem Skalarprodukt von Eins ist die Normale des Polygons direkt in die Lichtquelle gerichtet. Je näher der Wert sich gegen Null bewegt, desto kleiner wird der Winkel, in dem das Licht auf die Oberfläche des Polygons fällt. Der Lichteinfluss wird daher schwächer. Die Lichtberechnung während des Geometrieschritts hat jedoch einen entscheidenden Nachteil. Die Lichtinformationen abseits der Mittelpunkte der Polygone werden mittels Interpolation der umgebenden Polygone bestimmt. Die geometrische Struktur des Objekts wird so durch die Schattierung sichtbar (vergleiche Abbildung 2)[13].

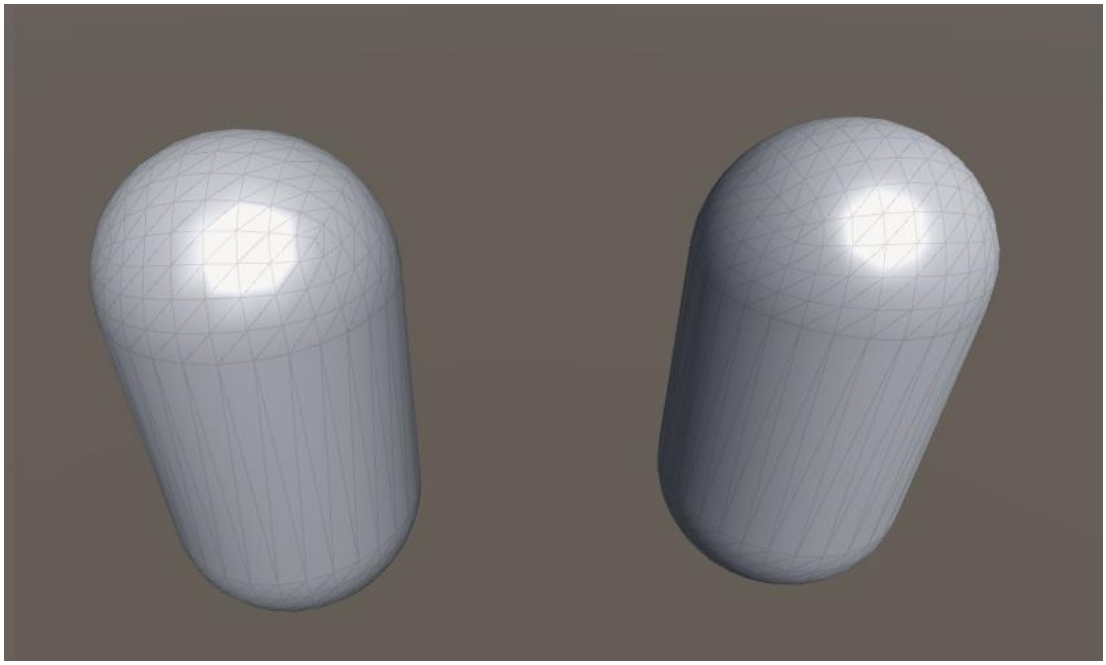

Abbildung 2: Lichtberechnung im Vertex Shader (linke Kapsel) umd im Pixel Shader (rechte Kapsel) mit Wireframe der Kapselgeometrie

Um die Qualität der Lichtberechnung zu erhöhen, kann die Lichtberechnung in der Forward-Renderingpipeline auch während des Rasterschritts durchgeführt werden. Dabei werden die Lichtinformationen für jeden Pixel des zu rendernden Bildes errechnet. Das optische Erscheinungsbild wirkt daher erheblich realistischer. Gleichwohl bleibt ein großer Nachteil der Forward-Renderingpipeline erhalten. Für mehrere Lichtquellen innerhalb einer Szene muss der entsprechende Shader für jeden Pixel genauso oft angewendet werden, wie es Lichtquellen gibt. Im ungünstigsten Fall ergibt sich die Komplexität daher aus der Anzahl der Objekte in der Szene multipliziert mit der Anzahl der Lichtquellen. Als Folge der gesteigerten Qualitätsansprüche setzt sich seit 2011 das Konzept der Deferred Renderingpipeline durch[4].

Deferred Rendering

Deferred Shading (engl. verzögerte Schattierung) betrachtet die Beleuchtung eine Szene als einen Bildeffekt, der auf den gerenderten Frame angewendet wird. Die Szene durchläuft dazu zunächst die klassische Renderpipeline bis zum Rasterschritt. Während des Rasterschritts werden jedoch zusätzliche Daten in den Framebuffer abgelegt, um die Lichtberechnung in einem anschließenden, zusätzlichen Schritt durchführen zu können. Die Anzahl der zusätzlichen Informationen variiert abhängig von dem gewünschten Effekte. Eine typische Aufteilung ist die Speicherung der Texturfarbinformationen in den RGB Kanälen und die Tiefeninformationen im Alphakanal des ersten Bilds. In einem zweiten Bild werden die Objektnormalen in den

drei Farbkanälen und die Specularity im Alphakanal abgelegt (vergleiche Abbildung 3)[4,27].

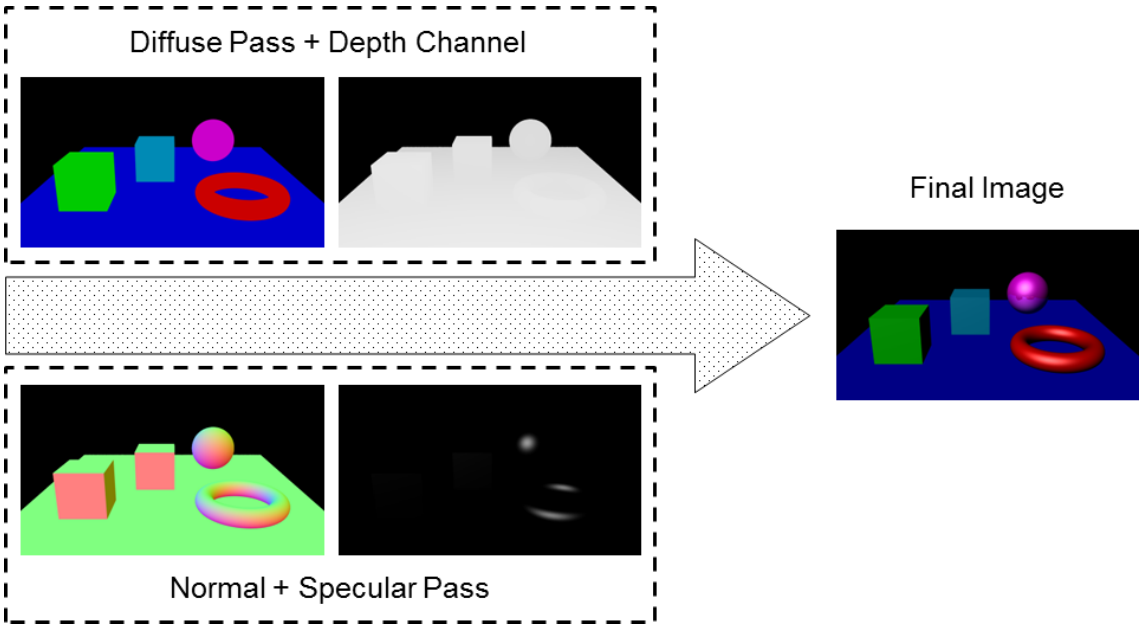

Abbildung 3: die zwei Framebuffer einer Defferd-Renderingpipeline und die resultierende Beleuchtung

Aus diesen Daten lassen sich die Lichtinformationen für jeden Pixel der Szene berechnen. Die Anzahl der dynamischen Lichtquellen in der Szene hat dabei einen geringen Einfluss auf die Berechnungsdauer. Ebenso können anstelle mehrerer großflächiger Lichter eine Vielzahl kleinerer Lichter eingesetzt werden, die so eine differenziertere Beleuchtung ermöglichen. Insbesondere bei Videospielkonsolen der letzten Generation, zum Beispiel Xbox 360 oder Playstation 3, wird Deferred-Rendering als gängige Rendertechnik eingesetzt.

Gleichzeitig muss bei diesem Verfahren eine Reihe von Bildeffekten und Materialeigenschaften anders verarbeitet und berechnet werden. Insbesondere transparente Oberflächen und Hardware Antialiasing sind von diesen Einschränkungen betroffen. Da die Lichtberechnung lediglich für jeden Pixel des finalen Frames durchgeführt wird, kann die Oberfläche zwar transparent sein, die dahinterliegenden Objekte jedoch nicht beleuchtet. Das gilt auch für Hardware gestützte Kantenglättung. Diese muss durch Postprocessing-Antialiasing ersetzt werden, die erst nach dem Rendervorgang über das Bild gelegt wird. Dazu muss zusätzlich ein Kantenerkennungsalgorithmus genutzt und dieser anschließend als Maske für einen Weichzeichner verwendet werden. Alternativ lässt sich auch Multisampling einsetzen, um einen ähnlichen Kantenglättungseffekt zu erzeugen[14]. Dieser Ansatz benötigt in der Praxis aber einen erheblich höhreren Rechenaufwand und wird primär auf leistungsstarken Systemen eingesetzt. Auch technische Probleme können durch den Einsatz einer Deferred-

Renderingpipeline entstehen, da die Framebuffer während des Rendervorgangs erheblich mehr Informationen übertragen müssen, als bei einer Forward-Renderingpipeline. Zusätzlich kann dieser Speicherauslastung durch komplexere Lichteffekte und Reflexionen noch gesteigert werden, da zusätzliche Framebuffer benötigt werden.

Abschließend lässt sich konstatieren, dass Deferred Rendering als alternatives Beleuchtungsverfahren an Bedeutung gewinnt und heute insbesondere in der interaktiven Unterhaltungsindustrie das dominante Beleuchtungsmodell darstellt.

2.3 Raytracing

Raytracing beschreibt ursprünglich Algorithmen, die mittels ausgesendeter Strahlen die Sichtbarkeit von Objekten in einem virtuellen Raum ermitteln. Dazu wird vom Auge des Betrachters der Szene ein Strahl durch die einzelnen Pixel der Bildebene geschickt. Es wird anschließend untersucht, welche Objekte die Bahn des Strahles schneiden und wie weit entfernt vom Ausgangspunkt des Strahles dieser Schnittpunkt liegt. Das Objekt, welches die geringste Entfernung zum Strahlursprung aufweist, ist von diesem Punkt aus sichtbar (vergleiche Abbildung 4). Diesen Vorgang nennt man Verdeckungsberechnung.

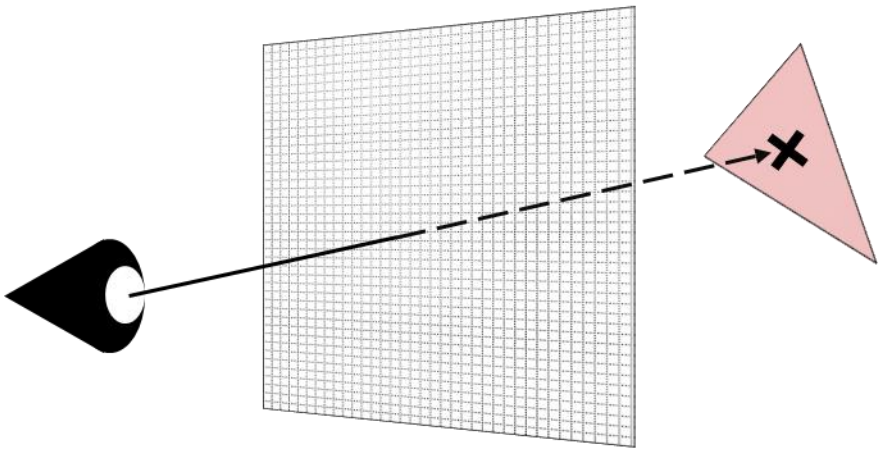

Abbildung 4: Projektion eines Polygons auf die Betrachterebene

Raytracing kann jedoch nicht nur genutzt werden, um Verdeckungsberechnung durchzuführen, sondern auch um das Verhalten von Lichtstrahlen in einer Szene zu simulieren[34]. Die erste Anwendung dieser Lichtberechnung mittels Raytracing ist die Berechnung von Schatten mittels eines Schattenstrahls. Dazu findet zunächst eine Verdeckungsberechnung mittels Raytracing statt. Von dem sichtbaren Punkt eines Objektes werden neue, sogenannte Schattenstrahlen in Richtung jeder Lichtquelle der Szene gesendet. Wenn diese Schattenstrahlen vor der Lichtquelle mit einem anderen

Objekt kollidieren, wirft dieses Objekt einen Schatten auf den sichtbaren Punkt. Wenn der Strahl ungehindert auf die Lichtquelle trifft, beleuchtet diese Lichtquelle direkt den sichtbaren Punkt und beeinflusst dessen Darstellung anhand der Materialeigenschaften des Objektes und den Lichteigenschaften der Lichtquelle (vergleiche Abbildung 5).

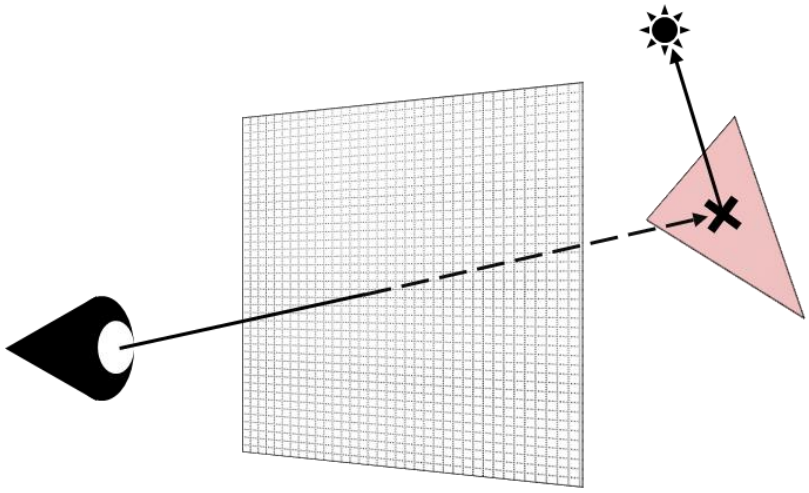

Abbildung 5: die Beleuchtung eines Polygons durch eine Lichtquelle

Bei der Schattenberechnung mittels Raytracing können jedoch keine durchsichtigen oder spiegelnden Oberflächen dargestellt werden. Daher wurde 1980 das Rekursive Raytracing von Key und Whitted entwickelt. Wenn nach der Verdeckungsberechnung eine spiegelnde Fläche dargestellt werden soll, wird ein weiterer Strahl entsprechend dem Reflexionsgesetz erzeugt. Das Reflexionsgesetz besagt, dass der Winkel, in dem ein Strahl durch die Reflexion zurückgeworfen wird, dem Eintrittswinkel entsprechen muss. Die erste Oberfläche, mit der ein Reflexionsstrahl kollidiert, beeinflusst zusätzlich mit seinen Materialeigenschaften das Aussehen des zu zeichnenden Punktes. Bei lichtdurchlässigen Objekten wird nach der Verdeckungsberechnung ein neuer Strahl anhand des Snellius‘schen Brechungsgesetzes in das Innere des zu zeichnenden Objektes gesendet. Da lichtdurchlässige Objekte das einfallende Licht auch zu einem gewissen Teil reflektieren, wird ein so genannter Sekundärstrahl erzeugt. Dieser folgt dem Reflexionsgesetz und wird entsprechend des Eintrittswinkels ausgesendet. Das Aussehen des lichtdurchlässigen Punktes wird anschließend anhand der Fresnel‘schen Formeln aus Bildinformationen der Lichtquellen, Schatten, Reflexionen und Brechungen zusammengesetzt.

Um ein möglichst realistisches Bild zu erzeugen, wird die Darstellung der Brechung und Reflexion rekursiv durchgeführt. Dazu werden von der Oberfläche, mit der der Reflexions- und Brechungsstrahl zuerst kollidiert, weitere Strahlen ausgesendet. Die eingestellte Berechnungstiefe legt dabei fest, wie oft dieser Vorgang rekursiv

durchgeführt wird. Moderne Raytracing Renderer nutzen einen Path Tracing Ansatz, um auch erweiterte optische Effekte wie weiche Schatten, glänzende Reflexionen und globale Beleuchtung zu errechnen. Globale Beleuchtung beschreibt die Simulation aller Wechselwirkungen zwischen Objekten einer Szene und dem Licht aller Lichtquellen im Raum. Die bekanntesten Wechselwirkungen sind Kaustiken und indirekte Beleuchtung. Raytracing wird in nahezu jedem Nichtechtzeitrenderer verwendet. Raytracing Verfahren erlauben es, mit steigender Renderzeit eine nahezu physikalisch korrekte Abbildung einer Szene zu erzeugen.

Raytracing ist primär ein Renderansatz für Nichtechtzeitgrafik, da die hohe Anzahl an Strahlen, die für eine hohe Bildqualität benötigt werden, derzeit nicht in Echtzeit verarbeitet werden können. Eine Reihe von kommerziellen Renderern nutzen jedoch GPU und CPU parallel, um innerhalb von wenigen Sekunden ein Bild mittels Raytracing zu berechnen.

2.4 Potentielle Einsatzszenarien für Rendersysteme

Die Einsatzgebiete für Rendersysteme sind vielfältig. Die Wahl zwischen den einzelnen Renderverfahren wird dabei maßgeblich von der zur Verfügung stehenden Renderzeit bestimmt. Echtzeitrendersysteme, wie sie in der Datenvisualisierung und in Videospielen zum Einsatz kommen, müssen ein Bild in möglichst kleiner Zeit errechnen. Daher werden für diese Aufgabe zumeist Polygonen-Renderer eingesetzt. Diese können von GPU gestützter Berechnungen profitieren und verzichten zu Gunsten einer schnellen Berechnung auf physikalisch korrekte Lichtdarstellungen.

Im Gegensatz dazu sind Raytracing-Render nur selten unter Echtzeitbedingungen realisierbar. Daher finden sie überwiegend in Produktionen Einsatz, bei denen die Bildberechnung und Darstellung zeitlich getrennt sind. Beispiele für diese Produktionen sind Produktvisualisierung, Computergrafik für Film, Fernsehen und Werbung sowie die Simulation komplexer Vorgänge unter möglichst realistischen Bedingungen. Moderne Raytracing-Renderer können ebenfalls Berechnungen auf die Grafikkarten der Recheneinheiten auslagern, erreichen jedoch zumeist keine Echtzeitbedingungen auf preisgünstigerer Hardware.

Es gibt zudem eine Vielzahl unterschiedlicher Rendersysteme, die schwerpunktmäßig für die Simulation von Prozessen eingesetzt werden. Ein prominenter Vertreter dieser Gruppe sind die Radiosity-Renderer, die ihr Einsatzgebiet vor allem in der Meteorologie finden.

3 Welche Herausforderungen stellt Paralleles Rendern dar?

Der Einsatz mehrerer Recheneinheiten gleichzeitig fordert immer eine Abwägung zwischen einer möglichst gleichmäßigen Verteilung und einem geringen Verteilungs- und Kommunikationsaufwand[18].

Eine gleichmäßige Verteilung aller Teilaufgaben auf die einzelnen Rendereinheiten bestimmt maßgeblich, wie effektiv die verfügbare Leistung genutzt wird. Im Idealfall müssen alle Recheneinheiten nahezu gleichgroße Aufgabenpakete erhalten, um gleichzeitig die einzelnen Teilaufgaben fertigstellen zu können. Sollten einzelne Recheneinheiten deutlich zeitaufwändigere Berechnungen durchführen, kann Leerlauf entstehen, welcher die Effizienz des gesamten Systems beeinträchtigt.

Dem gegenüber steht der Aufwand, die gleichmäßigen Teilaufgaben zu erzeugen. Dabei ist die Herausforderung, mit einem geringen Aufwand möglichst verlässlich vorherzusagen, wieviel Zeit einzelne Berechnungen benötigen. Dies kann jedoch auch bedeuten, dass Berechnungen, die eigentlich Bestandteil des Rendervorgangs sind, bereits während der Verteilung der Aufgaben durchgeführt werden müssen. Um diese möglichst zeitnah zu bearbeiten, müssen auch diese Berechnungen auf mehrere Recheneinheiten verteilt werden. Damit steigt der Kommunikationsaufwand.

Der Kommunikationsaufwand zwischen den Recheneinheiten kann beim parallelen Rendern einen zusätzlichen Flaschenhals bilden, da die Geschwindigkeit des Datenaustausches begrenzt ist.

3.1 Potentielle Hardware Szenarien für Paralleles Rendern

Je nach Einsatzgebiet und Art der Berechnungen ergeben sich eine Reihe von möglicher Hardware Szenarien für einen parallelen Rendervorgang. Zunächst können die Recheneinheiten eine Reihe von Prozessorkernen innerhalb eines einzelnen Computers sein (I). Es ist jedoch eher ungebräuchlich, dass ein Echtzeitrenderer ausschließlich die Hauptprozessoren eines Systems verwendet. In der Praxis werden große Teile der Berechnungen auf die Grafikprozessoren des Systems ausgelagert (II). Diese sind speziell für die Anforderungen einer grafischen Berechnung ausgelegt und können große Mengen mathematischer Berechnungen parallel ausführen. Gleichzeitig ist der Prozessor nach wie vor an den Berechnungen beteiligt. Dessen Vorteil liegt in der Flexibilität, unterschiedlichste Arten von Berechnungen durchzuführen. Das dritte

Einsatzszenarium ist die Kombination mehrere Prozessoren und/oder Grafikkarten über eine Netzwerkverbindung (III). Dazu wird eine Reihe von Computern mit einer Grafikkarten- Prozessor-Kombination über Netzwerktechnik verbunden. Nachfolgend werden diese Szenarien im Detail vorgestellt.

3.1.1 Multi-GPU Systeme

Die Ansteuerung eines Grafikprozessors erfolgt über eine Grafikschnittstelle und nicht direkt durch das entsprechende Programm. Damit werden die Möglichkeiten für eine Parallelisierung des Rendervorgangs durch die Grafikschnittstelle eingeschränkt. Der Leitungsgewinn im Vergleich zu Single-GPU Systemen ist daher abhängig vom Einsatzgebiet. Für Videospiele erreichen zwei identische Grafikkarten ca. 50% - 90% mehr Frames pro Sekunde als eine einzelne Grafikkarte[29]. Bei Anwendungen, die keine Grafikdarstellung über eine Grafikschnittstelle erzeugen sondern nur Berechnung auf die Grafikkarte auslagern, liegt der Leistungsgewinn zwischen 80% und 95%[30]. Diese Diskrepanz zwischen theoretischer Leistung und praktischem Mehrwert entsteht durch die unterschiedlichen Verteilungsansätze und die Art der Berechnungen.

Ein entscheidendes Nadelöhr ist dabei immer die Kommunikation zwischen den einzelnen Recheneinheiten. Die Grafikprozessoren sind dabei über die PCI Express Schnittstellen des Mainbords an das interne Bussystem angeschlossen. Diese Schnittstelle unterstützt in der aktuellen Version als PCI Express 3.0 x16 Standards eine Verbindungsrate von 15,75 GB/s zum Hauptprozessor des Systems. Der Datenaustausch ist daher mit einer vergleichsweise hohen Geschwindigkeit möglich.

Abhängig vom Hersteller der verwendeten Grafikkarten unterscheiden sich die physischen Schnittstellen zwischen den einzelnen Grafikprozessoren in einigen Details. NVIDIA verwendet die ursprünglich von 3dfx entwickelte Scan-Line-Interleave Technik, diese ist aktuell unter dem Namen Scalable-Link-Interface (SLI) bekannt[28]. ATI Technologies Lösung für Multi-GPU Systeme heißt CrossFire und wird heute unter dem Namen AMD CrossFireX eingesetzt[33]. Beide Techniken verwendeten ursprünglich eine zusätzliche Kabelverbindung zwischen den Grafikkarten mittels einer SLI oder CrossFireX Bridge. Diese Verbindung wurde parallel zu der bestehenden Verbindung über das Mainboard genutzt, um Daten direkt zwischen den Grafikprozessoren zu übertragen. Auf Grund der neueren PCI Express Schnittstellen verlieren diese direkten Verbindungen jedoch an Bedeutung. AMD CrossFireX überträgt lediglich 900MB/s und wurde daher zwischenzeitlich von der neueren XDMA Technik abgelöst. Diese benötigt keine zusätzliche Kabelverbindung. NVIDIA nutzt zum aktuellen Zeitpunkt weiterhin die SLI Bridge.

Eine besondere Form der Multi-GPU System ist die Verwendung des Integrated Graphics Processor (IGP) und der eigentlichen Grafikkarten des Computers. Dabei wird der bei vielen modernen Prozessoren enthaltene Grafikchip in die parallele Bearbeitung eines Rendervorgangs mit einbezogen. Dieser Grafikchip wird eigentlich genutzt, um den Stromverbrauch bei einfachen grafischen Anwendungen zu reduzieren, imdem die Hauptgrafikkarte ausgeschaltet wird. In der Praxis ist der Mehrwert dieser Hybrid CrossFireX Technologie jedoch eher gering und spielt daher häufig keine nennenswerte Rolle.

3.1.2 Netzwerk Rendersysteme

Für den Bereich Videospiele ist die Verwendung mehrerer Grafikkarten in einem System meistens ausreichend. Insbesondere, da sich Videospiele an den Endkonsumenten richten. Kommerzielle Echtzeitanwendungen zum Beispiel im Bereich Produktvisualisierung können jedoch weitaus mehr Ressourcen benötigen als Videospiele. Daher ist der nächste Schritt, um die Rechenleistung des Systems zu erhöhen, die Verwendung mehrerer Computer. Alle Einheiten sind dabei baulich getrennt und über ein Netzwerk verbunden. Diese Verbindung ist gleichzeitig die größte Schwachstelle. Die durchschnittliche Übertragungsrate innerhalb eines Netzwerkes liegt zwischen 100 Mbit/s und 10 Gbit/s. Diese Bandbreite muss zwischen allen Einheiten im Netzwerk aufgeteilt werden. Um die Bandbreite möglichst effektiv ausnutzen zu können, wird die Verteilung der Aufgaben und Daten zumeist durch eine zentrale Recheneinheit durchgeführt. Diese Einheit ist nicht zwangsläufig an der eigentlichen Berechnung der Frames beteiligt, führt jedoch sämtliche Verteilungsaufgaben und Datenübertragungen durch.

4 Verteilungsansätze

Das Hauptproblem eines parallelen Rendervorgangs liegt in der Unterteilung der Szene in kleinere Teilaufgaben, die jeweils von einer Recheneinheit bearbeitet werden können. Es bestehen drei grundsätzliche Verfahren, diese Unterteilung durchzuführen. Das Objektverteilungsverfahren unterteilt die zu rendernde Szene, in der die einzelnen Objekte in Objektgruppen zusammengefasst werden. Diese können anschließend auf die verfügbaren Recheneinheiten verteilt werden. Im Gegensatz dazu zerlegt das Bildverteilungsverfahren den zu rendernden Frame in einzelne Teilbilder. Diese können im Anschluss auf die Recheneinheiten gesendet werden[18]. Der dritte Verteilungsansatz zerlegt eine längere Sequenz in einzelne Frames und verteilt diese.

Nachfolgend werden diese Ansätze vorgestellt und hinsichtlich ihrer Vor- und Nachteile verglichen. Weiterhin wird Kohärenz als Möglichkeit betrachtet, aus vorangegangenen Berechnungen Schlüsse für ähnliche Arbeiten zu ziehen. Dies könnte sich positiv auf die Ressourcennutzung und Verteilung auswirken.

4.1 Kohärenz als Faktor der Parallelisierung

Kohärenz beschreibt in der Rendertechnik den Zusammenhang zwischen Elementen, die räumlich und/oder zeitlich nahe beieinander liegen und daher ähnliche Eigenschaften haben[18].

Kohärenz spielt in der Computergrafik eine große Rolle, da die Erfahrungen aus den vorangegangenen Berechnungen auf ähnlich gelagerte Sachverhalte übertragen werden können. Damit beschleunigt sich insgesamt der Prozess und verringert den Kommunikationsaufwand. Typische Vertreter für Elemente, zwischen denen Kohärenz auftreten kann, sind die Pixel eines Bildes, Strahlen eines Raycasters oder aufeinanderfolgende Frames einer Szene.

Frame-Kohärenz tritt zwischen zwei oder mehr Frames einer bewegten Szene auf. Teile des vorangegangenen Frames finden sich räumlich verschoben auch im nächsten Frame der Szene. Dabei müssen nicht unbedingt Pixeldaten aus dem vorangegangenen Frame in den aktuellen eingesetzt werden. Vielmehr wird versucht, die Berechnung des vorangegangenen Frames zu nutzen, um die aktuellen Berechnungen zu beschleunigen. Dieser Ansatz wird insbesondere bei den Strahlen eines Raycasters erkennbar. Für jeden Strahl muss bestimmt werden, welche Objekte den Strahl kreuzen und an welcher Position. Dazu muss ein Großteil aller in der Szene befindlichen Objekte einzeln überprüft werden. Wenn der Strahl in der Vergangenheit mit bestimmten Objekten kollidiert ist, können diese im nächsten Frame zuerst auf eine

Kollision mit dem neuen Strahl an der gleichen Stelle überprüft werden. Die Wahrscheinlichkeit ist hoch, dass diese Objekte erneut mit dem Strahl interagieren. Gleiches gilt auch für die Strahlen innerhalb eines einzelnen Frames. Werden mehrere Raycasts von einem ähnlichen Punkt aus in die Szene gesendet, interagieren diese mit hoher Wahrscheinlichkeit auch mit den gleichen Objekten in der Szene.

Da Kohärenz den Renderprozess grundsätzlich beschleunigen kann, ist zu prüfen, ob dieser Effekt auch beim parallelen Rendern über ein Netzwerk genutzt werden kann. Voraussetzung ist, dass im Netzwerk alle Recheneinheiten nützliche Daten der vorangegangenen Frames speichern und abrufen können. Um die Bandbreite der Netzwerkverbindung nicht zu belasten, müsste eine Vorauswahl aller Daten durchgeführt werden. Dazu müsste der Server alle Berechnungsdaten erhalten und die entsprechenden Teile an andere Recheneinheiten weiterleiten. Da der Verwaltungs- und Transportaufwand erheblich ist, kann nicht gewährleistet werden, dass die Daten der einzelnen Datenpakete rechtzeitig über das Netzwerk zu den anderen Recheneinheiten transportiert werden können. Während des Transports muss der Empfänger auf den Erhalt der Daten warten. In der Gesamtheit verlängert sich durch diesen „Leerlauf“ die Renderzeit im Netzwerk. In der Praxis wird sich die Nutzung der Kohärenz zwischen den Recheneinheiten daher als nicht praktikabel erweisen. Um Kohärenz im Netzwerk sinnvoll einzusetzen, sollte bereits bei der Verteilung der Aufgabenpakete berücksichtigt werden, ob innerhalb einer Teilaufgabe Kohärenz gegeben ist. Dabei wird der erhebliche Kommunikationsaufwand zwischen den einzelnen Recheneinheiten vermieden.

Kohärenz lässt sich aber nicht nur anwenden, um die Berechnungen während des Rendervorgangs zu optimieren, auch bei Datenverteilung und Speicherung lässt sich Kohärenz zwischen Datensätzen ausnutzen. So müssen viele Teile der Geometriedaten nicht für jeden Frame neu übertragen werden sondern können im Speicher behalten werden. Datenkohärenz sollte möglichst oft genutzt werden, da in vielen Rendersystemen die Datenverteilung zwischen den einzelnen Recheneinheiten eine mögliche Schwachstelle ist.

4.2 Die Funktionsweise der Objekt-Parallelisierung

Nach dem Anwendungsschritt liegt der Verteilungseinheit die Szene als Ansammlung von Objekten bestehend aus Vertexpunkten, Hilfsobjekten und Texturen vor. Diese Daten sollen nun möglichst gleichmäßig auf die einzelnen Recheneinheiten eines Rendersystems verteilt werden.

Um die Ansammlung von Objekten nun zu verteilen, muss zunächst die Komplexität jedes einzelnen Objektes bestimmt werden, um einschätzen zu können, wie lange

jedes einzelne Objekt für den Geometrieschritt verarbeitet werden muss. Da während des Geometrieschrittes jeder Vertexpunkt eines Objektes gleichermaßen verrechnet werden muss, ist die Komplexität eines Objektes direkt abhängig von der Anzahl der Vertexpunkte, die dieses Objekt bilden. Ist die Komplexität jedes Objektes in der Szene bekannt, müssen diese gleichmäßig verteilt werden. Da es in der Praxis meistens mehr Objekte in einer Szene gibt als Recheneinheiten in der Renderfarm, werden Objektgruppen gebildet. Nach dem Geometrieschritt erfolgt die Verarbeitung im Rasterschritt, in dem das finale Bild erzeugt wird. Dazu berechnet jede Recheneinheit die Bildinformationen, die durch die zuvor zugeteilten Objekte beeinflusst werden. Diese einzelnen Bildabschnitte werden anschließend von einer zentralen Einheit zusammengesetzt. Die Abbildung 6 zeigt diesen Vorgang für eine Beispielszene und deren Berechnung auf zwei getrennten Grafikkarten[16].

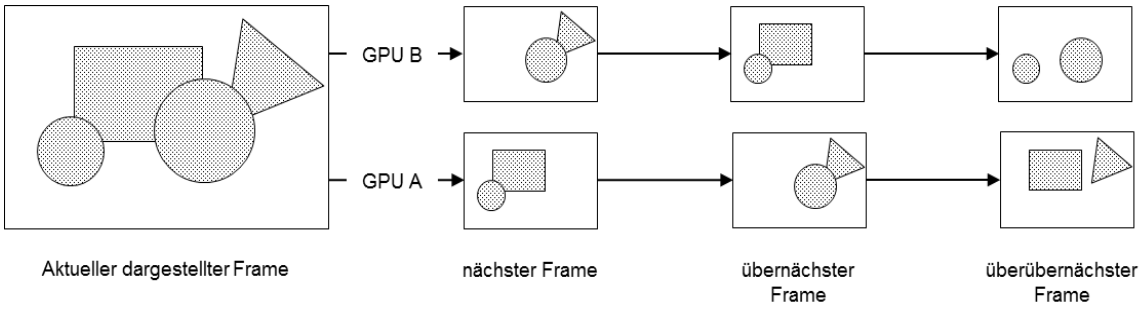

Abbildung 6: Schematische Darstellung eines Objektverteilungsverfahrens auf zwei Grafikkarten

4.3 Die Funktionsweise der Bild-Parallelisierung

Das Bild-Parallelisierungsverfahren unterteilt den zu rendernden Frame basierend auf dessen Abmessungen in Pixeln. Dazu wird die Fläche des zu rendernden Frames in kleinere Abschnitte zerteilt. Für jeden Abschnitt wird ermittelt, welche Objekte in diesem Bereich vorkommen. Für diese Pixel und Objekte werden die Berechnungen dann auf einer Recheneinheit vorgenommen. Die Abschnitte und ihre zugehörigen Objekte werden entsprechend der Anzahl der einzelnen Recheneinheiten in der Renderfarm gewählt. Die fertigen Abschnitte müssen nach dem Rendern nur noch zusammengesetzt und als einzelner Frame gespeichert werden.

Es existieren mehrere Möglichkeiten, die zu rendernde Fläche in kleinere Abschnitte zu zerlegen. Dabei wird zwischen dynamischen und statischen Verfahren unterschieden[19]. Dynamische Verfahren analysieren die Geometriedichte eines Abschnittes und errechnen daraus dessen Komplexität. Dynamische Verfahren können auch unterschiedlich leistungsstarke Recheneinheiten in die Arbeitsverteilung mit

einbeziehen und die Größe der Abschnitte anpassen. Dem stehen die statischen Unterteilungsverfahren entgegen. Dabei ist das Muster, nach dem der Frame unterteilt wird, immer gleich. Die Szene selbst muss nicht analysiert werden und hat keinen Einfluss auf die einzelnen Teilaufgaben. Ein typischer Vertreter für dieses Verfahren ist das Unterteilen nach einem Schachbrettmuster. Dabei wird das Bild in Zeilen und Spalten aufgeteilt und jede Recheneinheit erhält ein Feld aus dem Frame. Ein weiterer Vertreter ist das Scanline Verfahren, bei dem das Bild in horizontale oder vertikale Streifen unterteilt wird. Alle statischen Verteilungsverfahren versuchen möglichst von der Kohärenz zwischen Pixeln einer Gruppe zu profitieren. Daher wird in der Praxis davon abgesehen, Pixel, die sich räumlich nicht in unmittelbarer Nähe voneinander befinden, in einen Abschnitt zu platzieren.

Für Multi-GPU Systeme wird dieser Ansatz in Form des Split-Frame-Renderings (SFR) eingesetzt. Dabei wird jeder Frame entweder horizontal oder vertikal in zwei Teile zerlegt. Abbildung 7 zeigt ein horizontales Split-Frame-Rendering auf zwei unterschiedlichen Grafikkarten über den Zeitraum von vier Frames[16].

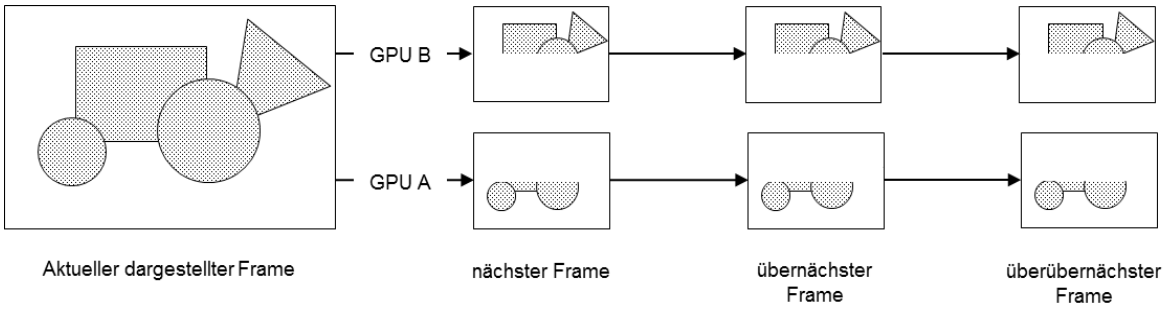

Abbildung 7: Schematische Darstellung eines Split-Frame-Renderings mit zwei Grafikkarten

4.4 Die Funktionsweise der Frame-Verteilung

Im Gegensatz zu den bisher vorgestellten Verteilungsansätzen lässt sich die Frame-Verteilung nur auf das Rendern einer Reihe von Frames anwenden. Die Parallelisierung eines einzelnen Frames ist aufgrund der Arbeitsweise der Verteilung nicht möglich.

Für die Parallelisierung wird die Reihe von Frames in kleinere Framegruppen zerlegt und auf die verfügbaren Recheneinheiten verteilt. Jede Recheneinheit bearbeitet diese völlig unabhängig von anderen Einheiten, kann dabei aber auch keine Kohärenz zwischen den Frames unterschiedlicher Einheiten nutzen. In der Praxis findet sich dieser Ansatz in den Alternate-Frame-Rendering Verfahren für Multi-GPU Systeme wieder[16]. Abbildung 8 stellt ein Alternate-Frame-Rendering exemplarisch mit zwei Grafikkarten dar.

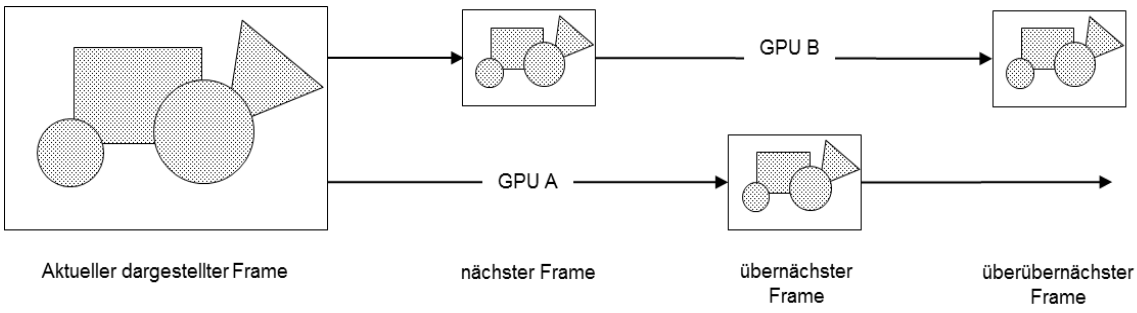

Abbildung 8: Schematische Darstellung eines Alternate-Frame-Rendering Prozesses auf zwei Grafikkarten

Bei dieser speziellen Form der Frame-Verteilung berechnen die verfügbaren Grafikeinheiten abwechselnd einzelne Frames einer Sequenz. Die anderen Grafikkarten des Systems bereiten in dieser Zeit die nächsten Frames vor. Dieses Verfahren erzeugt eine erheblich höhere Framerate als ein Single-GPU System. Gleichzeitig wird ein Frame immer noch nur von einer Recheneinheit bearbeitet, auch wenn sich diese abwechseln. Die Komplexität der Szene wird daher nach wie vor von der Leistung der einzelnen Grafikkarten bestimmt.

Weiterhin kann bei diesem Verfahren der Effekt der Mikroruckler auftreten. Sie entstehen, wenn sich zwei aufeinanderfolgende Frames signifikant in ihrem Berechnungsaufwand unterscheiden[26]. Die Berechnungsdauer des zweiten Frames ist daher erheblich größer als die des ersten Bildes. In einem System mit zwei Grafikeinheiten arbeitet die Grafikkarte, die den ersten Frame berechnet hat, zwischenzeitlich an dem dritten Frame. Da der zweite Frame vor dem dritten dargestellt werden muss, steht für den dritten Frame eine erheblich längere Berechnungszeit bereit. Diese unterschiedliche Wartezeit zwischen Frames einer Sequenz nimmt der Nutzer als eine Verlangsamung und anschließende Beschleunigung des dargestellten Szenarios wahr. Auf Grund der Arbeitsweise des AFR Verfahrens ist es nicht möglich, Mikroruckler grundsätzlich zu verhindern. Es besteht jedoch die Möglichkeit, das Auftreten dieser Ruckler in der Wahrnehmung des Nutzers zu reduzieren. Dazu wird die Ausgabe der Frames leicht verzögert. Erst wenn der nächste Frame einen bestimmten Bearbeitungspunkt überschritten hat, wird der voran gegangene Frame dargestellt. Die vergrößerte Wartezeit wird so auf mehrere kleine Wartezeiten aufgeteilt. Dabei wird aber vor ausgesetzt, dass die Anwendung mit einer hohen Bildwiederholungsfrequenz dargestellt wird, damit die Verzögerung zwischen der Benutzereingabe und der Darstellung nicht wahrnehmbar ist.

4.5 Vergleich dieser Ansätze

Jeder der oben beschriebenen Ansätze bringt eine Reihe von Vor- und Nachteilen mit sich. Die Hauptkriterien sind dabei die Komplexität des Verteilungsverfahrens, die gleichmäßige Verteilung im Geometrie- und Rasterschritt sowie der Kommunikationsaufwand durch die Verteilung.

Der Objektverteilungsansatz garantiert eine nahezu optimale Verteilung während des Geometrieschritts der Renderpipeline, da die Objekte in kleinere Gruppen mit nahezu gleicher Vertexpunkt Anzahl zusammengefasst werden. Gleichzeitig sagt die Anzahl der Vertexpunkte innerhalb eines Objekts nichts über dessen Größe und Anteil am zu rendernden Frame aus. Eine Objektgruppe im Hintergrund der Szene beeinflusst erheblich weniger Pixel als ein großes Objekt im Vordergrund mit einer geringen geometrischen Komplexität. Da der Berechnungsaufwand während des Rasterschritts kein Faktor für die Verteilung darstellt, kann es zu einer ungleichmäßigen Berechnungsdauer der einzelnen Teilaufgaben kommen. Dies reduziert die Effizienz des Rendersystems. Der Kommunikationsaufwand für ein Bildverteilungsverfahren setzt sich aus der Übertragung der Objekte für die Verteilung und den einzelnen Teilbildern nach dem Rendervorgang zusammen. Damit liegt der Kommunikationsaufwand auf einem mittleren Level. Die Implementierung und Komplexität der eigentlichen Verteilung ist für das Objektverteilungsverfahren sehr gering. Es müssen lediglich die Anzahl der Vertexpunkte innerhalb der einzelnen Objekte bestimmt werden. Das Zusammensetzen der erzeugten Teilbilder zum fertigen Frame hingegen benötigt zusätzliche Berechnungen, um die Sichtbarkeit jedes Rohpixels zu bestimmen. Dies erfolgt mittels einer Tiefenprüfung vor der Ausgabe des Bildes. Dieser Compositingschritt ist nicht Bestandteil der klassischen Renderpipeline und muss zusätzlich implantiert werden. Weiterhin kostet die erneute Tiefenprüfung der Szene zusätzliche Ressourcen und verschlechtert die Effizienz des Rendervorgangs.

Die Bildverteilungsverfahren hingegen optimieren vor allem die Verteilung während des Rasterschritts der Renderpipeline. Um einschätzen zu können, welche Objekte in einem bestimmten Bildabschnitt liegen, muss eine Vorberechnung durchgeführt werden. Dazu wird die Position jedes Objekts im zu rendernden Frame bestimmt. Dieser Vorgang benötigt zu Beginn des Rendervorgangs etwas Zeit. Gleichzeitig sind die Teilaufgaben für den aufwändigen Rasterschritt sehr gleichmäßig verteilt. In vielen modernen Anwendungen wird die Grafikkarte jedoch auch genutzt um verschiedene Berechnungen der Vertexpunkte durchzuführen. Dazu gehören das Skinning der Charaktere und Tessellation. Diese Berechnungen erfolgen während des Vertexshader der Renderpipeline. Für diesen sind die Objekte jedoch nicht optimal verteilt. Es kann zu ungleicher Verteilung und damit resultierenden Wartezeiten kommen. Der Kommunikationsaufwand für ein Bild-Verteilungsverfahren ist deutlich geringer als der

eines Objektverteilungsverfahrens, da bei diesem auch Pixel übertragen werden müssen, die durch andere Objekte verdeckt werden und somit nicht zum finalen Frame beitragen. Für ein Bild-Verteilungsverfahren müssen lediglich die Pixel übertragen werden, die auch wirklich dargestellt werden. Der zusätzliche Compositingschritt des Objekt-Verteilungsverfahrens entfällt. Gleichzeitig wird eine Vorberechnung notwendig, um noch vor dem Geometrieschritt erste Aussagen über den zu rendernden Frame treffen zu können. Der Implementierungs- und Berechnungsaufwand durch die Verteilung der Aufgaben ist daher vergleichbar mit denen des Objekt-Verteilungsverfahrens.

Die Frameverteilungssysteme zeichnen sich durch eine einfache und unkomplizierte Umsetzung aus. Die Renderpipeline bleibt unverändert. Dies ist von besonderer Bedeutung, wenn der Rendervorgang potentiell sowohl auf einer Recheneinheit als auch auf mehreren bearbeitet werden soll. Gleichzeitig findet die Verteilung unabhängig der Szenegeometrie oder des zu rendernden Frames statt. Die Verteilung kann daher nicht optimal erfolgen. Der Kommunikationsaufwand für dieses Verfahren ist sehr gering, da lediglich fertig berechnete Frames an die Ausgabeeinheit übertragen werden müssen. Eine Übertragung von Teilergebnissen ist nicht notwendig. Weiterhin finden keine Vorberechnungen oder zusätzliche Schritte während des Rendervorgangs statt, die die Renderzeit beeinflussen könnten.

Je nach Einsatzszenario eignet sich das Frameverteilungssystem vor allem für Rendervorgänge, bei denen paralleles Rendern lediglich eine zusätzliche Option ist und nicht zwingend benötigt wird. Sowohl Objekt- als auch Bildverteilung optimieren die Aufgabenverteilung erheblich besser als Frameverteilung. Beide haben aber auch deutliche Vor- und Nachteile. Eine Kombination beider Verteilungsverfahren wäre daher besonders vielversprechend.

5 Parallelisierungsverfahren und ihre Klassifizierung

Die im vorangestellten Kapitel gezeigten Ansätze finden sich in den meisten Verteilungsalgorithmen wieder. Um diese klassifizieren zu können, unterteilt S. Molnar die Verteilungsverfahren in „A Sorting Classification of Parallel Rendering“ in drei Gruppen[9]. Jeder Verteilungsansatz in diesen Gruppen besteht aus einer Vorverteilung zu Beginn der Renderpipeline und der Hauptverteilung, bei der teilweise Daten aus der Vorverteilung einbezogen werden. Die Gruppen unterscheiden sich daher sowohl anhand der Stelle in der Renderpipeline, an der die Verteilung stattfindet als auch in der Art der Daten, die während der Hauptverteilung zwischen den Recheneinheiten übertragen werden. Die klassische dreiteilige Renderpipeline wird auf eine zweiteilige Renderpipeline reduziert, da der Anwendungsschritt vergleichsweise wenig Ressourcen benötigt und einfach parallelisiert werden kann. Aus dieser zweiteiligen Pipeline ergeben sich drei mögliche Punkte der Hauptverteilung.

Die Gruppe der Sort-First-Verfahren errechnet während der Vorberechnung die Position jedes Objektes auf der zweidimensionalen Betrachterebene. Mit diesen Daten findet noch vor dem Geometrieschritt die Hauptverteilung mittels eines Bildverteilungsverfahrens statt. Diese erzeugten Teilaufgaben werden anschließend von derselben Recheneinheit im Geometrie- und Rasterschritt bearbeitet. Abbildung 9 zeigt einen Ausschnitt aus einer Renderpipeline mit einer Vorverteilung, Hauptverteilung und dem Geometrie- und Rasterschritt.

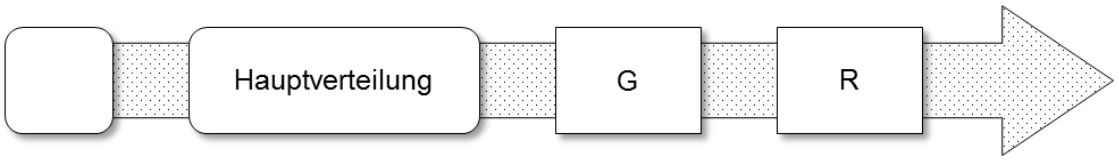

Abbildung 9: Teilabschnitt einer Renderpipeline mit einer Sort-First-Verteilung vor dem Geometrieschritt

Die Sort-Middle-Algorithmen führen ihre Hauptverteilung zwischen den beiden Schritten der Renderpipeline durch. Dazu werden die Objekte zunächst während der Vorverteilung grob auf die Recheneinheiten übertragen und im Geometrieschritt bearbeitet. Anschließend werden diese Daten genutzt, um die Objekte für den Rasterschritt erneut zu verteilen. Abbildung 10 zeigt die Arbeitsweise in einer Renderpipeline mit einer Vor- und Hauptverteilung.

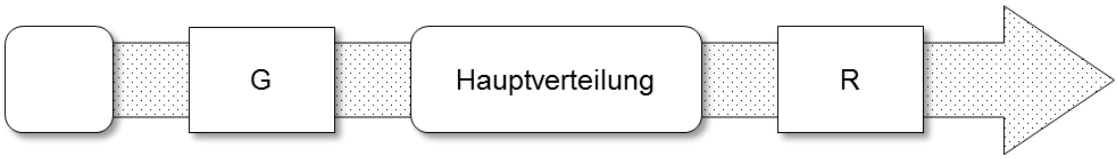

Abbildung 10: Teilabschnitt einer Renderpipeline mit einer Sort-Middle-Verteilung zwischen dem Geometrie- und Rasterschritt

Die Sort-Last-Algorithmen stellen die dritte Gruppe dar. Sie führen die Hauptverteilung nach dem Geometrie- und Rasterschritt durch. Damit eine Neuverteilung nach dem eigentlichen Berechnen der Szene realisierbar wird, muss ein zusätzlicher Schritt nach dem Rasterschritt eingeführt. Abbildung 11 zeigt eine Renderpipeline mit einer Vorverteilung zu Beginn der Renderpipeline. Nach dem Geometrie- und Rasterschritt schließt sich die Hauptverteilung mit Compositingschritt an.

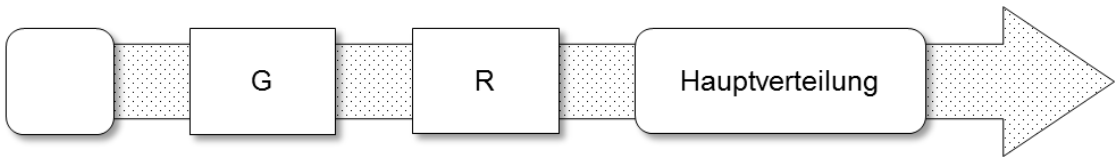

Abbildung 11: Teilabschnitt einer Renderpipeline mit einer Sort-Last-Verteilung nach dem Rasterschritt

Dieser Compositingschritt übernimmt Teile der Berechnungen des Rasterschritts und setzt aus den Pixelrohdaten der Recheneinheiten ein einzelnes Bild zusammen. Dazu ist es unter anderem nötig, eine Tiefenprüfung durchzuführen, um zu bestimmen, welche Pixel und die dazugehörigen Objekte andere Objekte verdecken und welcher Pixel damit dargestellt wird.

S. Molnar beschreibt dieses Problem als ein Sortierungsproblem[9]. Dabei muss festgestellt werden, welche Objekte innerhalb des Sichtbereiches liegen und daher gerendert werden und welche ignoriert werden können. Fehler in dieser Zuordnung würden zu einem sinnlosen Verbrauch von Ressourcen führen. Es ergibt sich daraus das Problem, möglichst genau zu bestimmen, welche Objekte innerhalb des Sichtfeldes der Kamera liegen, ohne zu viele Berechnungen vor der eigentlichen Verteilung durchzuführen.

Nachfolgend werden die drei Kategorien ausführlicher beschrieben und auf ihre Funktionsweise untersucht. Weiterhin werden die drei Gruppen verglichen und mögliche Vor- und Nachteile aufgezeigt.

5.1 Die Gruppe der Sort-First-Algorithmen und ihre allgemeine Arbeitsweise

Die Gruppe der Sort-First-Algorithmen ist eine Implementierung des Bildparallelisierungs-ansatzes. Dazu wird die Szene zunächst in kleinere Objektgruppen zerlegt[31]. Jede verfügbare Recheneinheit erhält ohne konkretes Verteilungssystem eine dieser Objektgruppen, um die sogenannte Pre-Transformation durchzuführen. Ziel dieser Vortransformation ist es, zu bestimmen, wo jedes Objekt auf der Bildebene liegt. Das Objekt wird dazu vom dreidimensionalen Raum der Szene in den zweidimensionalen Bildraum überführt. Dieser Vorgang ist jedoch mit erheblichem Rechenaufwand verbunden, da nahezu alle Berechnungsschritte des Geometrieschritts notwendig sind. Um die Vorberechnungen zu beschleunigen, wird zunächst für jedes Objekt eine Bounding-Box erzeugt. Eine Bounding-Box, auch Hüllkörper genannt, ist ein einfacher geometrischer Grundkörper, der einen komplexen räumlichen Körper vollständig umschließt. Je nach Anwendungsfall wird dazu meist ein Quader oder eine Kugel verwendet. Die erzeugte Bounding-Box wird anschließend anstelle des Objekts auf die Betrachterebene projiziert. Damit lässt sich abschätzen, wo ein Objekt ungefähr auf der Bildebene liegt, ohne komplexe Berechnungen durchführen zu müssen.

Das zu rendernde Bild wird anschließend in kleinere Teilbilder zerlegt. Jede Recheneinheit führt die für dieses Teilbild nötigen Berechnungen im Geometrie- und Rasterschritt durch. Sollte ein Objekt in mehreren Teilbildern liegen, werden die notwendigen Berechnungen auf jeder Recheneinheit durchgeführt, die Teile dieses Objekts darstellt. Das finale Bild muss am Ende des Rendervorgangs lediglich aus den Teilbildern zusammengesetzt und ausgegeben werden[1].

Eine potentielle Schwachstelle dieses Verfahrens ist die erste Verteilung für die Pre-Transformation, da große Teile der berechneten Daten für die Hauptverteilung erneut übertragen werden müssen. Es ist daher erstrebenswert, wenn die Vorverteilung bereits im Wesentlichen der späteren Hauptverteilung entspricht. Wenn lediglich ein einzelner Frame einer Szene gerendert werden soll, lässt sich dies nur mit einem erheblichen Mehraufwand erreichen. Wenn jedoch mehrere aufeinanderfolgende Frames einer Szene gerendert werden sollen, wie dies bei interaktiven Anwendungen der Fall ist, kann Framekohärenz genutzt werden, um aus der Erfahrung der vorangegangenen Frames Rückschlüsse auf noch kommende Berechnungen zu ziehen[1]. Dazu wird der zuletzt gerenderte Frame untersucht und bestimmt, welche Objekte in welchem Berechnungsabschnitt liegen. Basierend darauf wird die Vorverteilung so gestaltet, dass möglichst viele Objekte während der Pre-Transformation von der gleichen Recheneinheit verarbeitet werden wie später im

Geometrie-und Rasterschritt. Dies spart Zeit und Kommunikationsaufwand zwischen den Einheiten. Gleichzeitig lässt sich die Framekohärenz auch nutzen, um die Berechnungszeiten der einzelnen Teilschritte vorherzusagen. Mit hoher Wahrscheinlichkeit werden die gleichen Teilaufgaben für den aktuellen Frame eine ähnliche Berechnungsdauer benötigen wie im vorangegangenen Frame. Damit lassen sich Leerlaufzeiten reduzieren und zuverlässige Aussagen über die wahrscheinliche Berechnungszeit treffen.

Bewertung der Sort-First-Algorithmen

Der große Vorteil der Sort-First-Algorithmen ist der geringe Kommunikationsaufwand während der zwei Verteilungsvorgänge. Für die erste Verteilung müssen lediglich die einzelnen Objekte auf die Recheneinheiten verteilt werden. Die Daten aus den Vorberechnungen werden nicht aktiv für den Geometrie- und Rasterschritt genutzt und müssen daher nicht übertragen werden. Daher müssen für die Hauptverteilung nur die Objektdaten verteilt werden, die nicht bereits für die Vorberechnungen auf die richtige Recheneinheit gesendet wurden. Wenn Framekohärenz genutzt werden kann, sinkt die Anzahl der übertragenen Daten weiter und benötigt den geringsten Kommunikationsaufwand aller Verteilungsverfahren.

Ein weiterer Vorteil ist, dass die klassische Renderpipeline weitestgehend ohne Veränderungen genutzt werden kann. Daher eignen sich Sort-First-Algorithmen besonders für Systeme, bei denen paralleles Rendern als eine Option implementiert werden soll. Weiterhin können bestehende Systeme oder externe Bibliotheken parallelisiert werden, ohne weitreichenden Zugang zum eigentlichen Renderer zu besitzen.

Gleichzeitig sind die Sort-First-Algorithmen anfällig für eine ungleiche Verteilung und bieten selten die optimale Ausnutzung der verfügbaren Ressourcen. Da Objekte in mehreren Teilaufgaben gleichzeitig liegen können, kann es zu Berechnungsredundanz kommen. Weiterhin tendieren Objekte innerhalb einer Szene dazu, Anhäufungen und kleine Gruppen zu bilden und damit Teilaufgaben mit deutlich unterschiedlichen Berechnungszeiten zu generieren.

Zwar kann die Nutzung von Framekohärenz einige der beschriebenen Nachteile abschwächen, setzt jedoch selbst ein aufwändiges Verwaltungssystem voraus. Dieses wird benötigt, um die Informationen der vorangegangenen Frames zentral zu sammeln und auszuwerten. Damit steigt der Aufwand während der Implementierung und bindet Teile der verfügbaren Rechenleistung. Dieser Aufwand steht dem geringen Implementierungs-aufwand des eigentlichen Verteilungssystems entgegen.

Steven Molnar merkt in seiner Klassifizierung jedoch an, dass zum Zeitpunkt seiner Arbeit der Nutzen eines Sort-First-Algorithmus fraglich ist[9]. Da die Komplexität der Berechnungen seither erheblich gestiegen ist, lassen sich Sort-First-Algorithmen deutlich effizienter einsetzen.

Abschließend lässt sich festhalten, dass die Gruppe der Sort-First-Algorithmen sich besonders für Anwendungen eignet, bei denen bereits bestehende Renderer parallelisiert werden sollen. Es ist mittels Sort-First sogar möglich, eine optionale Parallelisierung in einen Renderer zu implementieren, da diese Algorithmen die Renderpipeline weitestgehend unberührt lassen. Weiterhin eignen sich diese Verteilungsverfahren für Anwendungen, bei denen eine überschaubare Kommunikation zwischen den Recheneinheiten stattfindet.

5.2 Die Gruppe der Sort-Middle-Algorithmen und ihre allgemeine Arbeitsweise

Sort-Middle-Algorithmen führen die Hauptverteilung zwischen dem Geometrie- und Rasterschritt durch. Dazu werden für den Geometrieschritt zunächst alle Objekte gleichmäßig auf die einzelnen Recheneinheiten des Systems verteilt. Ausschlaggebendes Kriterium für diese erste Verteilung kann die geometrische Komplexität des Objekts sein. Diese lässt sich auf die Anzahl der Vertexpunkte innerhalb des Objektes vereinfachen. Dieser Verteilungsvorgang benötigt besonders wenige Ressourcen, da viele Dateiformate die Anzahl der Vertexpunkte speichern und daher lediglich abgerufen werden müssen.

Nach den Berechnungen im Geometrieschritt findet eine weitere Verteilung für den Rasterschritt statt. Diese nutzt ein Bildverteilungsverfahren, um neue Teilaufgaben zu bilden. Alle Daten, die für die Berechnung der einzelnen Pixelgruppen aus dem Geometrieschritt benötigt werden, müssen über die Verbindung der einzelnen Recheneinheiten ausgetauscht werden. Die im Rasterschritt erzeugten Teilbilder werden anschließend zusammengesetzt und ausgegeben.

Bewertung der Sort-Middle-Algorithmen

Die Sort-Middle-Algorithmen zeichnen sich durch eine Verteilung der Aufgaben an einer sehr praktischen Stelle in der Renderpipeline aus. Insbesondere, wenn die Geometrie- und Rastereinheiten getrennt voneinander sind, bietet sich eine Neuverteilung an dieser Stelle aus technischer Sicht an. Theoretisch lässt sich an dieser Stelle eine nahezu optimale Verteilung für den Rasterschritt erzeugen. Der Geometrieschritt lässt sich in der Vorverteilung ebenfalls nahezu optimal in

Teilaufgaben zerlegen. Damit bieten die Sort-Middle-Verfahren die beste Verteilung, wenn lediglich eine gleichmäßige Verteilung gefordert ist. Gleichzeitig erfordert dieses hohe Maß an Neuverteilung auch einen großen Kommunikationsaufwand, da nahezu alle Daten aus dem Geometrieschritt erneut übertragen werden müssen. Bei Szenen mit einer hohen Tessellationsrate und vielen komplexen Objekten, kann dieser Übertragungsaufwand den Rendervorgang verlangsamen und einen Flaschenhals bilden[1].

Damit sind die Sort-Middle-Verfahren für Anwendungen geeignet, die eine möglichst optimale Verteilung auf die verfügbaren Ressourcen benötigen. Die Kommunikation innerhalb des Systems muss vergleichsweise schnell und koordiniert ablaufen. Dies spricht daher für eine zentrale Steuereinheit, die den Rendervorgang überwacht und die Datenübertragung steuert. Einsatzszenarien mit getrenntem Raster- und Geometrieschritt eignen sich daher ideal für die Sort-Middle-Algorithmen. Gleiches gilt für große und leistungsstarke Rendersysteme, die speziell für paralleles Rendern konzipiert sind.

5.3 Die Gruppe der Sort-Last-Algorithmen und ihre allgemeine Arbeitsweise

Die Gruppe der Sort-Last-Algorithmen ist eine Umsetzung der Objekt-Parallelisierung. Die erste Verteilung findet zu Beginn des Geometrieschritt statt und verteilt die Objekte gleichmäßig auf die verfügbaren Recheneinheiten. Diese Verteilung bleibt sowohl für den Geometrieschritt als auch für den Rasterschritt erhalten. Nach dem Rasterschritt liegen für jedes Objekt die entsprechenden Pixel vor, die von diesem Objekt beeinflusst werden. Diese müssen in einem zusätzlichen Compositingschritt zu einem Bild zusammengesetzt werden. Für diesen Compositingschritt findet eine zweite Verteilung der Aufgaben statt. Dabei wird der zu rendernde Frame in kleinere Teilbilder zerlegt und den Recheneinheiten für den Compositingschritt zugewiesen[1].

Zentrales Alleinstellungsmerkmal der Sort-Last-Verteilungsverfahren ist der zusätzliche Compositingschritt. Dieser wird nötig, da die einzelnen Objekte zwar auf ihre Pixel reduziert werden, dabei aber nicht geprüft werden kann, ob dieses Objekt sichtbar oder verdeckt ist. Diese Tiefenprüfung scheitert daran, dass nicht jedes Objekt auf jeder Recheneinheit verfügbar ist. Es kann daher auch kein einheitlicherer Tiefenkanal generiert werden. Ähnliches gilt für reflektierende und/oder durchsichtige Oberflächen, die unabhängig von der Tiefenprüfung durch eine Vielzahl von Objekten beeinflusst werden. Dazu müssen zunächst alle Pixelrohdaten über die Verbindung der Recheneinheiten an die Compositingeinheiten übertragen werden. Dies kann über zwei unterschiedliche Arten geschehen.

Bei dem Sort-Last-Sparse Ansatz werden lediglich die Pixel übertragen, die tatsächlich auf der entsprechenden Recheneinheit erarbeitet wurden. Dabei ist der Datenaustausch vergleichsweise gering. Gleichzeitig müssen die einzelnen Pixel später aufwändig in den Kontext des zu rendernden Frames gesetzt werden. Dies führt in der Regel zu einem verlängerten Rendervorgang führen.

Der Sort-Last-Full-Frame Ansatz platziert alle errechneten Pixel in einem Bild und füllt die von anderen Recheneinheiten bearbeiteten Pixel mit transparenten Bildpunkten auf. Diese Bilder können im Compositingschritt zeitnah zum finalen Frame zusammengesetzt werden. Gleichzeitig erhöht der Full-Frame Ansatz den Kommunikationsaufwand und kann zu einer Flaschenhalsbildung in der Datenübertragung führen.

Bewertung der Sort-Last-Algorithmen

Ein großer Vorteil der Sort-Last-Algorithmen ist, dass sie eine nahezu unveränderte Renderpipeline verwenden. Mit Ausnahme des Compositingschrittes ist die Implementierung daher vergleichbar mit einem Renderer ohne Parallelisierung. Lediglich der Compositingschritt übernimmt Teilaufgaben des Rasterschritts und muss zusätzlich implementiert werden[7].

Da zu Beginn der Renderpipeline in der Vorverteilung die Anzahl und Komplexität der Objekte genutzt wird, ist die Verteilung sehr gleichmäßig und wenig anfällig für Berechnungsredundanz. Weiterhin sinkt die Wahrscheinlichkeit für einen Leerlauf einzelner Recheneinheiten drastisch. Gleichzeitig stellen die späte Hauptverteilung und der Einsatz eines Compositingschrittes aber die größte Schwachstelle dieses Verfahrens dar. Während der Hauptverteilung müssen alle Pixelrohdaten über die Verbindung der Recheneinheiten erneut übertragen werden. Im Gegensatz zu anderen Verfahren werden auch Pixeldaten übertragen, die später nicht dargestellt werden müssen, da sie von Objekten anderer Recheneinheiten verdeckt werden. Der Kommunikationsaufwand ist daher erheblich höher als bei Verfahren, die lediglich Teilbilder des zu rendernden Frames übertragen.

Insbesondere bei Anwendungen, die Multisampling einsetzen, kann der Kommunikationsaufwand rapide ansteigen[1]. Multisampling berechnet für einen später sichtbaren Pixel eine Vielzahl von Subpixeln, die individuell berechnet werden müssen. Für die Kantenglättung (Antialiasing) werden diese Subpixel anschließend auf einen einzelnen Pixel reduziert. Da Antialiasing besonders an Kanten zwischen Objekten auftritt, steigt die Wahrscheinlichkeit, dass diese Pixel von unterschiedlichen Recheneinheiten bearbeitet werden. Damit erhöht sich neben dem Aufwand im Compositingschritt auch der Aufwand in der Datenverteilung. Gleiches gilt für

Downsampling, bei der der zu rendernde Frame in einer erheblich höheren Auflösung, als später dargestellt, gerendert wird. Das Skalieren dieses höheraufgelösten Bildes glättet ebenfalls die Kanten. Beide angesprochenen Verfahren sind in der Praxis besonders bei aufwändigen Anwendungen üblich.

Abschließend ist festzustellen, dass sich Sort-Last-Verfahren besonders für den Einsatz in Systemen mit einer guten Anbindung der Recheneinheiten untereinander eignet. Multi- und Downsampling sollten zugunsten anderer Antialiasingmethoden jedoch nur selten oder überhaupt nicht eingesetzt werden.

5.4 Der Hybrid-Sort-First-Sort-Last-Algorithmus

Als Schlussfolgerung aus der Bewertung der drei Algorithmengruppen ergibt sich, dass keines der Verfahren eine ideale Verteilung bei geringem Kommunikationsaufwand erreicht. Ein logischer Schritt ist daher, mehrere Verfahren und Algorithmen zu kombinieren, um die Nachteile zu kompensieren. Die Wissenschaftler der Princton University Rudrajit Samanta, Thomas Funkhouser, Kai Li und Jaswinder Pal Singh nutzen diesen Ansatz und kombinieren den Sort-First-Algorithmus mit Elementen einer Sort-Last-Verteilung[32]. Dieser Hybrid-Ansatz stellt eine vielversprechende Alternative zu den bisher beschriebenen Verteilungsmöglichkeiten dar. Bei dieser Verteilung kommen sowohl Elemente der Objekt- als auch der Bild-Parallelisierung zum Einsatz. Dazu wird der Rendervorgang in drei Phasen unterteilt.

In der ersten Phase findet die Verteilung der Aufgaben auf die verfügbaren Recheneinheiten statt. In der zweiten Phase werden die verteilten Aufgaben sowohl im Geometrie- als auch Rasterschritt bearbeitet. In der dritten und letzten Phase des Renderprozesses wird der „finale Frame“ aus den einzelnen Teilbildern zusammengesetzt. Nachfolgende Abbildung stellt die Phasen für die Berechnung einer Beispielszene grafisch dar.

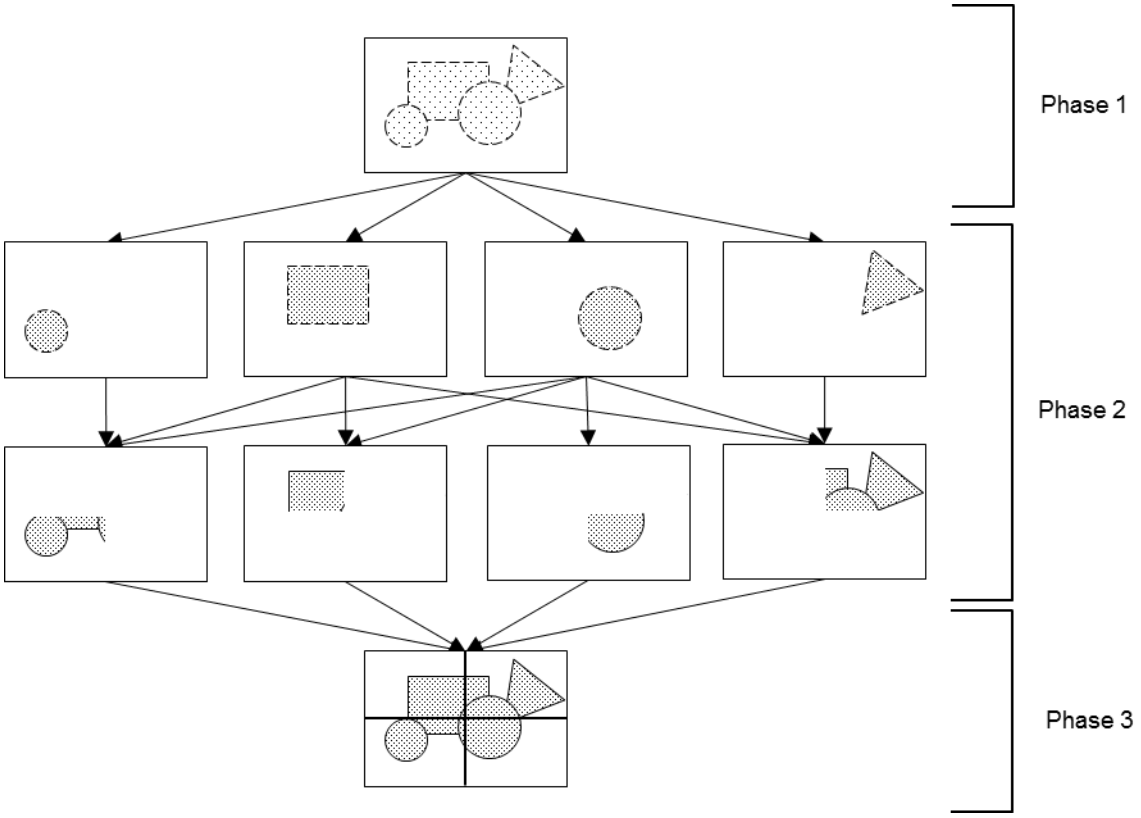

Abbildung 12: Schematische Darstellung der drei Bearbeitungsphasen eines Hybrid Verteilungsverfahrens

Die erste Phase beginnt, ähnlich der der Sort-First-Algorithmen, mit der Pre-Transformation der einzelnen Objekte. Dabei wird die Position und Größe jedes Objektes auf der Betrachterebene mittels einer Boundingbox bestimmt. Anschließend werden die beiden kürzesten, gegenüberliegenden Seiten des zu rendernden Frames bestimmt. Dabei ist es zwingend erforderlich, dass die Ausmaße des Bildes eine rechteckige oder auch quadratische Grundform aufweisen. Ausgehend von diesen beiden Seiten wird das Bild nun in zwei Bereiche unterteilt. Dazu „wandern“ pixelweise von den beiden Seiten jeweils Linien zur gegenüberliegenden Seite des Bildes und analysieren, welche Objekte sich innerhalb ihres Bereiches befinden. Wenn ein Objekt vollständig innerhalb eines der beiden Bereiche liegt, wird es diesem zugeordnet. Abbildung 13 zeigt diesen Vorgang in drei aufeinander folgenden Momentaufnahmen.

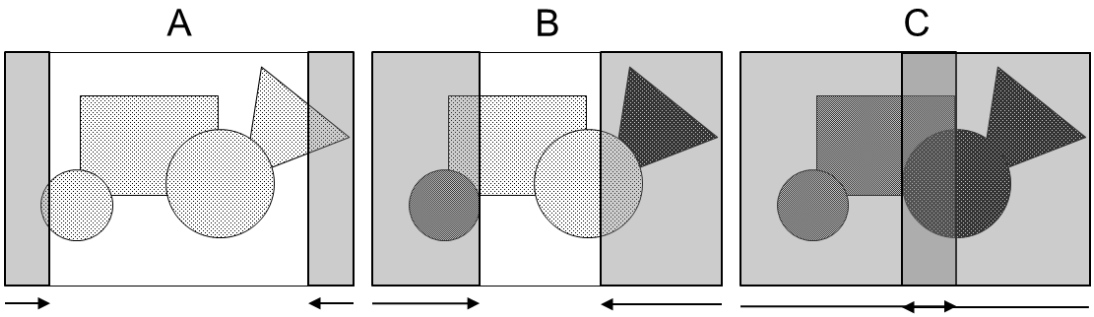

Abbildung 13: drei Momentaufnahmen auf dem Verteilungsvorgangs eines Hybrid-Verteilungsverfahrens

Die beiden grauen Bereiche des ersten Bildes zeigen die zwei bereits betrachteten Bereiche des Bildes. Bisher befindet sich noch kein Objekt vollständig in einem der beiden Bereiche (vergleiche Abbildung 13 Bild A). Die weiße Fläche stellt den Bereich dar, der noch nicht analysiert wurde. In Abbildung 13 Bild B sind bereits zwei Objekte den entsprechenden Teilbereichen zugeordnet. Das Dreieck wurde dem rechten Bildabschnitt zugeordnet. Der Kreis befindet sich im anderen Bereich. Bild 13 C stellt die fertige Verteilung der Objekte dar. Zum Zeitpunkt, an dem sich die beiden Bereiche in der Mitte des Bildes treffen, liegen noch zwei Objekte nicht vollständig in einem der beiden Bereiche. Daher wird die Analyse jeweils in den Bereich des anderen Bildabschnitts fortgesetzt. Es entsteht ein Bereich, der Objekte beider Teilbereiche enthält, hier dunkelgrau markiert. Die Pixel, die in diesem Bereich liegen, werden von Objekten beider Teilaufgaben beeinflusst und müssen ausgetauscht oder redundant berechnet werden. In diesem Fall werden sie der Recheneinheit zugewiesen, die bisher am wenigsten Objekte berechnen muss. Dieser Verteilungsvorgang wird nun rekursiv auf die beiden neuerzeugten Teilabschnitte angewendet, um weitere kleinere Teilaufgaben zu erzeugen. Insbesondere, wenn die Anzahl der Recheneinheiten einer Zweierpotenz entspricht, ist die Verteilung sehr gleichmäßig möglich. Jedes Teilbild wird am Ende der ersten Phase einer Recheneinheit für den Geometrie- und Rasterschritt zugewiesen.

In der zweiten Phase findet der wesentliche Rendervorgang während des Hybrid-Verteilungsverfahrens statt. Im Geometrieschritt werden alle Objekte, die in den einzelnen Teilbereichen liegen, auf Vertexebene bearbeitet. Anschließend bearbeitet die gleiche Recheneinheit diesen Bildabschnitt im Rasterschritt. Objekte, die teilweise oder vollständig in einer der Überschneidungszonen liegen, werden im Geometrieschritt von einer Recheneinheit bearbeitet. Die Daten werden anschließend zu jeder Recheneinheit gesendet, die dieses Objekt ebenfalls in ihrem Bildabschnitt vorfindet. Im Gegensatz zu anderen Verteilungsverfahren wird bereits während der Hauptverteilung festgelegt, welche Recheneinheit die mehrfach benötigten Daten errechnet und welche anderen Einheiten diese Daten benötigen. Daher kann im

Idealfall auf eine zentrale Steuereinheit komplett verzichtet werden. Am Ende der zweiten Phase werden die fertigen Teilbilder an die Compositingeinheit übertragen.

Die dritte Phase umfasst das Zusammensetzten der einzelnen Teilbilder zu einem fertigen Frame. Der Großteil der Bildinhalte kann gemäß des Verteilungsmusters, ähnlich eines Puzzles, zusammengebaut werden. Die Bildabschnitte, die Objekte mehrerer Teilaufgaben enthalten, durchlaufen einen aufwändigeren Compositingschritt. Dabei werden die unterschiedlichen Informationsteile ähnlich eines Sort-Last-Verfahrens verarbeitet. Der Frame wird anschließend zur Ausgabe in den Backbuffer der Anzeigeeinheit abgelegt und bei Bedarf angezeigt.

Im Gegensatz zu einem reinen Bild-Parallelisierungsverfahren betrachtet dieser Algorithmus nicht nur die Pixel eines Teilbereiches, sondern auch die Anzahl und Größe der einzelnen Objekte innerhalb dieses Teilbereiches. Damit verbindet dieser Ansatz den geringen Kommunikationsaufwand eines Sort-First-Algorithmus mit einer gleichmäßigen Verteilung auf alle Recheneinheiten. Im Gegensatz zu einem Sort-Middle-Algorithmus findet jedoch keine Neuverteilung zwischen dem Geometrie- und Rasterschritt statt. Der Datenaustausch erfolgt basierend auf den Berechnungen der Verteilung vor dem Rendervorgang. Hervorzuheben ist, dass die Sammlung der Zwischenergebnisse in einer zentralen Einheit nicht notwendig ist und damit der Kommunikationsaufwand deutlich verringert wird.

Laut einer Untersuchung der Princton University erreicht der beschriebene Hybrid-Ansatz eine drei- bis viermal höhere Effizienz im Vergleich zu einem reinen Sort-First-Algorithmus und eine zwischen 30 und 50 Prozent höhere Effizienz gegenüber einem Sort-Last-Verfahren.

Zu beachten ist, dass das in der Literatur beschriebene Verfahren voraussetzt, dass alle Objekte innerhalb eines Frames, geometrisch gesehen, eine gleiche Komplexität aufweisen. Theoretisch ist es möglich, dass alle Objekte innerhalb einer Szene gleich komplex sind. In der Praxis muss diese Voraussetzung bei der Erstellung einer Szene berücksichtigt werden. Bei Videospielen ist es üblich, große Objekte in kleinere Teilobjekte zu zerlegen, um Instancing zu benutzten. Diese Unterteilung muss verstärkt werden, um die Szene in noch kleinere Teile zu zerlegen. Zu analysieren wäre auch, ab welchem prozentualen Anteil ungleichmäßig komplexer Objekte an der gesamten Szene das Hybrid-Verfahren einer effizienten Verteilung entgegensteht.

Weiterhin ist es notwendig, dass die Objekte gleichmäßig in der Szene verteilt sind. Große Ansammlungen von Objekten können Teilabschnitte provozieren, in denen besonders viele Berechnungen notwendig sind. Damit wäre die Verteilung empfindlich gestört.

5.5 Zusatz Verteilung in einem Hybrid Sort-First-Sort-Last-Verfahren

Ähnlich dem Bild Parallelisierungsansatz bezieht das Hybrid-Verteilungsverfahren die Anzahl der Vertexpunkte pro Objekte nicht in die Verteilung der Aufgaben mit ein. Dies könnte einen Nachteil darstellen, da die Objekte innerhalb der Szene eine unterschiedliche Komplexität aufweisen und damit innerhalb des Geometrieschritts ungleich verteilt werden. Um eine gleichmäßigere Verteilung zu gewährleisten, sollte nach der Hybrid-Verteilung eine zusätzliche Überarbeitung der Teilaufgaben auf der Basis der geometrischen Komplexität erfolgen. Nachfolgend wird eine mögliche Erweiterung des Hybrid-Verfahrens vorgestellt.

Die Optimierung der Verteilung beginnt zunächst damit, dass für jedes Objekt der Szene die Anzahl der Vertexpunkte bestimmt und gespeichert wird. Die geometrische Komplexität eines Objektes wird maßgeblich von der Anzahl der Vertexpunkte bestimmt. Anschließend wird der Hybrid-Verteilungsalgorithmus wie gewohnt angewendet und das zu rendernde Bild in kleine Teilabschnitte zerlegt. Danach setzt die Optimierung der einzelnen Teilaufgaben auf Basis der Vertexpunkte ein. Dazu wird mit der zuvor bestimmten Anzahl der Vertexpunkte pro Objekt bestimmt, wie viele Vertexpunkte durchschnittlich innerhalb eines Teilbildes und deren Vertexpunkteanzahl bestimmt, wie viele Vertexpunkte insgesamt in jedem Teilabschnitt liegen. Aus der Summe der Vertexpunkte aller Teilabschnitte wird anschließend das arithmetische Mittel bestimmt. Wenn die Anzahl der Punkte außerhalb des Mittels liegt, muss eine weitere Anpassung der Objektverteilung in Betracht bezogen werden. Um den Kommunikationsaufwand für diese Verteilung zu minimieren, werden lediglich Objekte verteilt, die innerhalb des Überschneidungsbereiches zwischen den Teilschritten liegen. Diese Objekte müssten zwischen dem Geometrie- und Rasterschritt verteilungsbedingt auf andere Recheneinheiten übertragen werden, damit diese die Betrachtung auf Pixelebene durchführen können.

In der Praxis ist es jedoch nahezu unmöglich, dass die Summe aller Vertexpunkte einer Teilaufgabe exakt der durchschnittlichen Vertexpunkteanzahl entspricht. Daher wird ein Toleranzkorridor um das arithmetische Mittel gebildet. Alle Teilaufgaben mit einer Vertexpunktanzahl innerhalb des Korridors werden nicht erneut verteilt. Der Korridor kann für jede Anwendung und Systemkonfiguration individuell festgelegt werden. In einem Multi-GPU System kann auf Grund der hohen Übertragungsbandbreite ein kleinerer Toleranzwert genutzt und damit eine gleichmäßige Verteilung der Objekte ermöglicht werden. In einem Netzwerk muss der Toleranzwert deutlich höher angesetzt werden, um den Kommunikationsaufwand gering zu halten.

6 Umsetzung eines Hybrid Verteilungsverfahren an einem Beispiel

Abschließend zur Betrachtung der unterschiedlichen Verteilungsverfahren, folgt eine Implementierung eines Hybrid-Verteilungsverfahrens in einem Beispiel. Dazu wird ein bestehender Lineclipping Algorithmus auf mehrere Threads parallelisiert, um eine Vielzahl von Recheneinheiten zu simulieren. Das Ausgangsprogramm wurde ursprünglich von Prof. Haenselmann entwickelt. Nachfolgend wird zunächst die Ausgangslage betrachtet und analysiert, wie das Lineclipping-Programm funktioniert. Anschließend folgt die eigentliche Implementierung des Hybrid-Verteilungsverfahrens. Zu guter Letzt wird die parallelisierte Version dem ursprünglichen Programm gegenüber gestellt, um ein Fazit aus praktischer Sicht ziehen zu können.

6.1 Ausgangssituation

Die Implementierung des Hybrid-Verteilungsverfahrens erfolgt mittels C++ und der SDL Bibliothek in der Version 1.2 für Windows. Zusätzlich kommt die SDL_gfx Erweiterung zum Einsatz, um die Zeichenfunktionalität der SDL Bibliothek zu ergänzen.

Das der Implementierung zugrundeliegende Lineclipping-Projekt stellt ein Objekt als Wireframe Model dar. Dabei werden die sichtbaren Kanten (Edges) des Objektes weiß dargestellt. Alle nicht sichtbaren Linien werden für die perspektivische Darstellung ausgeblendet. Der Nutzer kann das sichtbare Objekt mittels gedrückter Maustaste rotieren, um den Blickwinkel anzupassen.

Die Hauptfunktionalität des Programmes liegt in zwei Funktionen. Die DisplayModel Funktion verarbeitet die Szene für jeden zu rendernden Frame und unterscheidet zwischen sichtbaren, teilweise sichtbaren und verdeckten Linien. Diese werden anschließend über die *DrawLine* Funktion mittels der SDL Bibliothek dargestellt. Die *DisplayModel* Funktion wird über eine Schleife in der main Funktion für jeden Frame aufgerufen. Die main Funktion verarbeitet darüber hinaus die Nutzereingaben und leert den Screen-Buffer vor dem nächsten Frame. Um diese Funktionen in der main.cpp gruppieren sich eine Reihe von zusätzlichen Klassen, um zum Beispiel die datentechnische Repräsentation eines Vektors im zwei- und dreidimensionalen Raum zu gewährleisten.

Die Speicherung aller Objekte innerhalb der Szene erfolgt durch ein zentrales Array vom Datentyp *Face*. Dieser beinhaltet die Ursprungspunkt, die eigentlichen Punktkoordinaten, die Polygonen-Normale und den Ursprungspunkt der Normalen.

Innerhalb des Faces Arrays wird nicht gespeichert, zu welchem Objekt ein bestimmtes Face gehört. Dies muss für eine Hybrid-Verteilung angepasst werden.

6.2 Entwicklung des Verteilungsverfahrens und Implementierung in das bestehende Programm

Als erster Schritt der Umsetzung des Hybrid-Verteilungsverfahrens, muss die Datenstruktur des Programms an die geänderten Anforderungen angepasst werden. Da viele Verteilungsverfahren die Objekte einer Szene verteilen, muss das Faces Array durch einen mit 3D Objekten gefüllten Vector ersetzt werden. Dazu wurde ein neues Struct mit der Bezeichnung Object eingeführt. Dieser beinhaltet einen Vector mit einer unbestimmten Anzahl Faces pro Objekt. Die Begrenzung der maximale Anzahl der Faces pro Objekt wurde damit aufgehoben. Neben dem Objekt selbst werden noch zusätzlich eine Reihe von weiteren Daten für die Verteilung gespeichert. Die Integer *currentGroup* gibt während der Verteilung an, ob dieses Objekt bereits innerhalb einer Verteilungsgruppe liegt oder bisher noch nicht betrachtet wurde. Zusätzlich geben vier Pointer die Außenpunkte einer rechteckigen Boundingbox an, die das Objekt auf der Bildebene vollständig umschließt. Diese Außenpunkte werden jedoch nicht als neue Punkte innerhalb des Objekts abgelegt, sondern referenzieren den Punkt des betrachteten Objekts, der in dieser Richtung am weitesten vom Mittelpunkt des Objektes entfernt liegt.

```
struct Object
{
    vector<Face> faces;
    int currentGroup;

    VectorR3* leftBoundingPoint;
    VectorR3* rightBoundingPoint;

    VectorR3* topBoundingPoint;
    VectorR3* bottomBoundingPoint;
};
```

Abbildung 14: Quelltext des Object Structs

Diese Veränderung der Datenstruktur macht es notwendig, dass anders als bisher, jedes Face innerhalb des Programms durch zwei Indizes adressiert werden muss. Der Objekt Index gibt dabei an, welches Objekt innerhalb des Objekt Vectors betrachtet wird. Der Face Index bestimmt, welches Face innerhalb eines bestimmten Objekts analysiert werden soll. Dies spielt insbesondere bei der Speicherung der Objekte eine Rolle, die eine gegebene Edge verdecken.

Um die *DisplayModel* Funktion an die geänderten Strukturen anzupassen, wurde der Schleifendurchlauf des Faces Arrays durch zwei ineinander verschachtelte Schleifen ersetzt, die für jedes Objekt im Array jedes Face aufrufen und die nötigen Berechnungen durchführen. Weiterhin mussten die Aufrufe und Vergleiche an die neue Struktur angeglichen werden.

Als letzter Teilschritt mussten die Funktionen angepasst werden, die neue Objekte zur bestehenden Szene hinzufügen. Stellvertretend dafür steht die Funktion *AddObjectCube*. Sie nutzt eine angepasste Version der *AddRectangle* Funktion, um das neue Objekt an einer bisher ungenutzten Stelle des Object Arrays zu platzieren. Die bisher genutzten Funktionen zur Objekterzeugung wurden deaktiviert, um fehlerhafte Daten im Object Array zu vermeiden.

Im zweiten Schritt der Umsetzung, folgt die Implementierung der parallelisierten *DisplayModel* Funktion. Dazu legt eine Konstante die maximale Anzahl der durch Threads simulierten Recheneinheiten fest.. Für jeden Thread muss ein *SDL_Thread* Pointer in einem Array abgelegt werden. Über diesen Pointer kann der entsprechende Thread später angesteuert werden.

Die *DisplayModel* Funktion nutzte bisher eine Reihe von Parametern, um mit der *main* Funktion kommunizieren zu können. Diese muss für den Einsatz in einem Thread in ein Struct gebündelt werden, da die Übergabe mehrerer Parameter an einen *SDL_Thread* nicht möglich ist.

```
typedef struct
{
    SDL_Surface* screen;
    VectorR3 observer;
    VectorR3 target;
    int threadColor;

    Object *objectIndex[NO_OBJECTS];
    int objectNumber;
} ThreadData;
////////////////////////////////////////
```

Abbildung 15: ThreadData Struct für den Datenaustausch zwischen der main Funktion und dem Thread

Neben den bisher eingesetzten Pointer auf die *SDL_Surface*, den *observer* Vektor und den *target* Vektor, kommen eine Reihe zusätzlicher Daten zum Einsatz. Die *threadColor* färbt alle Linien, die dieser Thread zeichnet, in einer besonderen Farbe ein. Damit lässt sich das Verteilungsverfahren während der Ausgabe visualisieren. Das Integer Array *objectIndex* speichert Pointer zu allen Objekten, die von dieser Recheneinheit bearbeitet werden sollen. Der *objectCount* legt fest, wie viele Objekte

tatsächlich von diesem Thread bearbeitet werden sollen und welche Einträge im *objectIndex* Array nur Leerwerte sind (vergleiche Abbildung 15).

Innerhalb der *DisplayModel* Funktion werden diese Daten in lokale Variablen übertragen und stehen dem Thread zur Verfügung. Um die unterschiedlichen Thread Daten innerhalb der *main* Funktion abzulegen, wird erneut ein Array aus *ThreadData* Structs genutzt.

Die eigentliche Erzeugung und Verwaltung der Threads findet in der *main* Funktion des Programms statt. Dazu wird für jeden zu rendernden Frame zunächst das *ThreadData* Array initialisiert. Anschließend findet die Verteilung der Teilaufgaben statt. Dazu wird das *objectIndex* Array mit den Indizes der Objekte der einzelnen Teilaufgaben befüllt. Anschließend werden alle Threads über die Funktion *SDL_CreateThread* erzeugt.

Um Darstellungsfehler zu verhindern, wird für jeden Thread anschließend der Befehl *SDL_WaitThread* ausgeführt. Durch diesen Befehl wartet das Programm darauf, dass der Thread alle ihm zugewiesenen Aufgaben erfüllt hat und das Programm ohne Risiko fortgesetzt werden kann.

Im dritten Schritt findet die Implementierung des Hybrid-Verteilungsverfahrens statt. Der Hybrid-Ansatz benötigt zu Beginn des Rendervorgangs eine Pre-Transformation, bei der die Boundingboxes für jedes Objekt erstellt werden. Diese Vorberechnungen werden ebenfalls parallel auf mehreren Einheiten ausgeführt. Erst nach diesen Berechnungen kann der Lineclipping-Algorithmus ausgeführt werden.

Die Pre-Transformation wird durch eine angepasste Version der *RotateObject* Funktion durchgeführt. Da auch diese Pre-Transformation über mehrere Threads parallelisiert werden soll, wurden die eingehenden Parameter zu einem Struct zusammengefasst. Dieses enthält neben dem *observer* und *gaze* Vektor ebenfalls ein Array mit den Indizes aller Objekte, die von diesem Thread bearbeitet werden sollen. Der Vektor *gaze* wird im Gegensatz zur ursprünglichen Funktion bereits vor der Pre-Transformation innerhalb der *main* Funktion berechnet, da dieser für alle Threads gleich ist.

```
typedef struct
{
    VectorR3 observer;
    VectorR3 gaze;

    int objectIndex[NO_OBJECTS];
    int objectCount;
} PreTransfromThreadData;
```

Abbildung 16: PreTransformThreadDate für den Datenaustausch zwischen main Funktion und den Pre-Transformation Threads

Um die Pre-Transformation durchführen zu könne, muss eine einfache Verteilung speziell für die Vorberechnungen erzeugt werden. Im vorliegenden Programm werden alle Objekte aufgerufen und abwechselnd einem Thread zugewiesen. Die räumliche Position der Objekte wird dabei nicht betrachtet.

Nach der Vorberechnung erfolgt die Aufgabenverteilung mittels des Hybrid-Sort-First-Sort-Last-Verfahrens. Dazu wird zunächst für jedes Objekt eine Boundingbox erstellt. Die dazu notwendigen Berechnungen werden durch die *SetBoundingBox* Funktion durchgeführt.

Die Boundingbox wird für jedes Objekt unabhängig erzeugt. Vier Pointer referenzieren zunächst den allerersten Punkt innerhalb der ersten Fläche. Dieser dient als Ausgangspunkt für die Vergleiche mit allen anderen Punkten innerhalb des Objektes. Anschließend wird jedes Face durchlaufen und für jeden Punkt bestimmt, ob er außerhalb der bisher errechneten Boundingbox liegt. Dies geschieht über einen Vergleich der Koordinaten mit dem bisher am weitesten außenliegenden Punkt. Sollte dies der Fall sein, wird die Boundingbox vergrößert um auch den gerade betrachteten Punkt einzuschließen. Am Ende der Betrachtung aller Punkte und Flächen innerhalb eines Objektes, referenzieren die vier Pointer die Punkte, die am weitesten außenliegenden und sind damit gleichbedeutend mit den Ausmaßen der Boundingbox.

Als letzter Schritt folgt die Verteilung der Aufgaben durch den Hybrid-Verteilungsansatz. Dies übernimmt die *HybridSubTaskMaker* Funktion. Von zwei Seiten wird pixelweise die Boundingbox jedes Objektes in der Szene verglichen und das Objekt in eine der beiden Objektgruppen platziert. Dies wird durch mehrere ineinander verschachtelte Schleifen realisiert.

Die Hauptschleife wird für die gesamte Breite des Teilbildes pixelweise durchlaufen. Darin wird die Boundingbox jedes Objektes mit der aktuell betrachteten Pixelreihe verglichen. Ein Objekt, welches mit zumindest einer Seite der Boundingbox innerhalb des Analysebereichs liegt, wird markiert. Diese Markierung dient dazu, dass ein Objekt

auch wirklich nur einer Objektgruppe zugewiesen werden kann und keine Objekte betrachtet werden, die bereits verteilt wurden. Diese Daten müssen innerhalb der Funktion gespeichert werden. Dazu wird ein Array mit Integer Werten namens inGroup genutzt. Jeder Eintrag im Array steht für ein einzelnes Objekt der Szene. Null steht dabei für ein bisher noch nicht betrachtetes Objekt. Eins und zwei geben an, dass dieses Objekt zumindest teilweise in einer der beiden Teilaufgabe liegt. Wenn ein Objekt vollständig innerhalb einer Teilaufgabe liegt, wird der *inGroup* Wert auf drei gesetzt und die Analyse dieses Objekts abgebrochen. Dieses Array wird jedoch nicht zur Kommunikation zwischen dem Hybridverteilungsverfahren und der *main* Funktion genutzt. Diese Aufgabe übernimmt die globale Variable *workGroups*. In ihr wird für jeden Thread ein Array mit allen zu bearbeitenden Objekten gespeichert. Diese Daten können anschließend in die ThreadData für die parallel laufenden Threads übertragen werden. Sollte die Anzahl der parallelen Recheneinheiten zwei überschreiten, muss der Hybrid-Verteilungsansatz mehrfach durchlaufen werden. Dies erfolgt durch den rekursiven Aufruf der Funktion *HybridSubTaskMaker*. Nachfolgend wird die Erzeugung der Teilaufgaben am Beispiel eines verteilten Rendervorgangs auf vier Recheneinheiten beschrieben.

```
void HybridSubTaskMaker(int screenRegionX, int screenRegionY,
                        int screenRegionWidth, int screenRegionHeight,
                        int currentRecursiveLevel, int currentScreenRegionNumber,
                        vector<Object*> objectGroup)
```

Abbildung 17: Funktion zur Erstellung der Teilaufgaben HybridSubTaskMaker

Zunächst wird der Funktion ein Vector mit den Indizes aller Objekte der Szene übergeben. Zusätzlich dazu wird die Größe des zu betrachteten Bildabschnitts und dessen Startkoordinaten übertragen. Diese werden genutzt um die beiden kürzesten Seiten des Bildes zu bestimmen. Dazu wird die Höhe des Bildes mit der Breite verglichen. Anschließend wird das Bild pixelweise von den beiden kürzeren Seiten in zwei Arbeitsgruppen unterteilt. Da im vorliegenden Beispiel Arbeitsgruppen für vier verfügbare Recheneinheiten erstellt werden sollen, wird die Funktion rekursiv noch einmal für die beiden erzeugten Teilgruppen aufgerufen. Abschließend werden die Daten aus den Arbeitsgruppen in das *workGroups* Array übertragen und sind damit bereit für den Rendervorgang.

Nach der Verteilung der Aufgaben werden die *workGroups* in die einzelnen Thread Datensätze übertragen. Für jede Teilaufgabe wird anschließend ein neuer *SDL_Thread* gestartet und mit einem Point referenziert. Nachdem alle Threads gestartet wurden, wird erneut ein *SDL_WaitThread* ausgeführt, um alle Berechnungen abzuschließen, bevor ein neuer Frame gezeichnet wird.

Die Pre-Transformation und der eigentliche Hybrid-Verteilungsansatz werden für jeden Frame aufgerufen und eine neue Verteilung speziell für das zu rendernde Bild erstellt. Kohärenz wird in dieser Implementierung nicht eingesetzt, da die berechneten Inhalte eine vergleichsweise geringe Komplexität aufweisen. Ein zusätzliches System für die Nutzung von Framekohärenz würde den Rendervorgang zusätzlich verlangsamen.

6.3 Fazit der Implementierung

Ein Vergleich zwischen den beiden Programmversionen zeigt, dass die überarbeitete Programmversion mit dem integrierten Hybrid-Verfahren eine erheblich höhere Framerate erreicht als die Ursprüngliche. Daraus lässt sich die These ableiten, dass der Hybrid-Verteilungsansatz den Rendervorgang grundsätzlich beschleunigt.

Zu beachten ist jedoch, dass die Implementierung des Hybrid-Verteilungsansatzes nur sinnhaft ist, wenn mehrere physisch getrennte Recheneinheiten zum Einsatz kommen. Liegen diese nicht vor, muss zumindest eine Simulation mehrerer Recheneinheiten erfolgen. In der Praxis ist diese bloße Simulation mehrerer Recheneinheiten wenig gewinnbringend, da die Analyse der Szene und deren Verteilung Ressourcen des Systems binden, die ansonsten dem Rendervorgang zur Verfügung gestanden hätten.

Die trotzdem zu beobachtende Beschleunigung des Rendervorgangs beruht im vorliegenden Fall darin, dass das ursprüngliche Programm leidglich 20 Prozent der im System verfügbaren Leistung arbeitete. Als Testsystem kam ein Intel Core i7 2600K unter Standardtakt zum Einsatz. Die Verwendung mehrerer Threads steigerte die genutzte Leistung auf ca. 60 Prozent.

Eine weitere These aus den Beobachtungen der praktischen Umsetzung des Hybrid-Verteilungsverfahrens ist, dass der Verteilungsaufwand, verglichen mit den Berechnungen für jeden Frame, einen erheblichen Mehraufwand darstellt. Dies ist aber nur für Anwendungen zutreffen, die vergleichsweise wenige Berechnungen für die Darstellung eines Frames durchführen müssen. Das in dieser Arbeit parallelisierte Lineclipping macht in den meisten 3D Visualisierungen nur einen geringen Anteil an den notwendigen Berechnungen aus. In einem typischen 3D Renderer wäre der Verteilungsalgorithmus bei der Leistungsbetrachtung zu vernachlässigen. Dies minimiert trotz alle dem natürlich nicht die Bestrebung, mit einem möglichst geringen Berechnungsaufwand eine nahezu optimale Verteilung zu erreichen.

Abschließend lässt sich für die praktische Implementierung des Hybrid-Verteilungsverfahrens festhalten, dass der Einsatz eines solchen Systems in vielen Fällen möglich und sinnvoll ist. Die Implementierung, insbesondere des Hybrid-Verteilungsverfahrens, ist mit einem überschaubaren zeitlichen und technischen Aufwand verbunden. Der erzielte Geschwindigkeitszuwachs rechtfertigt die zusätzlichen Arbeiten in den meisten Anwendungsfällen. Dies gilt im Besonderen für Anwendungen, die einen hohen Berechnungsaufwand erfordern, und Netzwerkrendersysteme.

7 Fazit

Multi-GPU Systeme und Netzwerkrendersysteme sind der Schlüssel zur Erzeugung komplexer und realistischer Bilder sowohl im Bereich der Echtzeitgrafik als auch Nichtechtzeitgrafik. Entscheidend für die möglichst effiziente Nutzung dieser Rendersysteme ist der Einsatz eines geeigneten Verteilungsverfahrens, um den Rendervorgang in kleine, von einer Recheneinheit bearbeitbare, Aufgabenpakete zu zerlegen. Dabei muss stets die Balance zwischen einer möglichst gleichmäßigen und zugleich rechentechnisch einfachen Verteilung gefunden werden. Im Fall der Netzwerkrendersysteme kommt zusätzlich noch eine Einschränkung in Form der verfügbaren Bandbreite dazu. In den unterschiedlichen Anwendungsgebieten der Rendersysteme haben sich im Laufe der Zeit verschiedene Verteilungsverfahren etabliert. In der interaktiven Unterhaltungsindustrie wird vor allem das Alternate-Frame-Rendering eingesetzt. In der Forschung und Produktvisualierung hingegen sind eine Vielzahl verschiedener Verfahren im Einsatz.

Im Verlauf dieser Arbeit zeigt sich, dass keine eindeutige Empfehlung für ein spezielles Verteilungsverfahren abgeben werden kann. Jeder Ansatz bringt eine Vielzahl an Vor- und Nachteilen mit sich. Es ist jedoch festhalten, dass sich die Verfahren als vielversprechend erweisen, die ein zu renderndes Bild sowohl auf der Objekt- als auch Bildebene betrachten. Stellvertretend für diese Gruppe wurde der Hybrid-Sort-First-Sort-Last-Verfahren vorgestellt. Dieses eignet sich durch den breiten Ansatz vor allem für Aufgabenstellungen, bei denen komplexe Berechnungen sowohl im Geometrie- als auch Rasterschritt durchgeführt werden. Dies ist bei vielen modernen Anwendungen, zum Beispiel Videospiele, der Fall, da neben der eigentlichen Bildberechnung auch viele Berechnungen aus den Bereichen Physik und Simulation auf die Grafikprozessoren ausgelagert werden.

Es hat sich jedoch auch gezeigt, dass die bisher genutzten Sort-Last- und Sort-Middle-Verfahren in spezialisierten Anwendungsbereichen ebenfalls sehr vielversprechende Ergebnisse produzieren. Die Entscheidung über das geeignete Verfahren muss daher für jede Anwendung individuell betrachtet werden. Maßgebliche Faktoren für Auswahl des Verfahrens können die Anzahl und Verteilung der Objekte, die verfügbare Bandbreite zwischen den Einheiten und die angestrebte Framezahl pro Sekunde sein.

Die Sort-First-Verfahren eigenen sich vor allem für Anwendungen, bei denen ein geringer Kommunikationsaufwand zwischen den Recheneinheiten angestrebt wird. Gleichzeitig geht dieses Verfahren aber zulasten einer gleichförmigen Verteilung und kann zu Leerlauf auf einzelnen Recheneinheiten führen.

Die Sort-Last-Verfahren hingegen benötigen einen zusätzlichen Compositingschritt und transportieren große Mengen an Pixelinformationen über die Datenverbindungen. Gleichzeitig nutzt dieses Verfahren eine nahezu unveränderte Renderpipeline. Dies senkt den Aufwand einer Implementierung deutlich und erhöht die Skalierbarkeit des Rendersystems.

Demgegenüber erreicht das Sort-Middle-Verfahren eine sehr gleichmäßige Verteilung zwischen den einzelnen Recheneinheiten. Dies hat zur Folge, dass der Verteilungsaufwand und die benötigte Bandbreite zunehmen.

Abschließend lässt sich festhalten, dass es grundsätzlich sinnvoll ist, mehrere Recheneinheiten in die Berechnung eines Bildes einzubeziehen. Dabei können die zusätzlichen Berechnungseinheiten nicht nur für die eine erhöhte Framerate eingesetzt werden, sondern auch zu einer erhöhten Bildqualität beitragen. Es gibt eine Vielzahl von Verfahren, die für die meisten Anwendungsgebiete eine passende Lösung darstellen. In der Gesamtabwägung stellt der Hybrid-Sort-First-Sort-Last- Algorithmus von allen in dieser Arbeit vorgeschlagenen Verfahren das vielversprechendste Verteilungssystem dar. Einen verstärkten künftigen Einsatz, insbesondere im Bereich der Unterhaltungsindustrie, wäre daher sinnvoll.

Literaturverzeichnis

[1] Carl Mueller. The Sort-First Rendering Architecture of High-Performance Graphics. I3D ’95 Proceedings of the 1995 symposium on Interactive 3D Graphics, 1995, 75-ff

[2] K. Debattista, V. Sundstedt, F. Pereira, A. Chalmers. Selective Parallel Rendering for High-Fidelity Graphics. Proceedings of Theory and Practice of Computer Graphics 2005, 2005, 59-66

[4] Shawn Hargreaves. Deferred Shading. Game Developer Conference 2004, 2004

[5] Stefan Eilemann, Renato Pajarola. Direct Send Compositing for Parallel Sort-Last Rendering. Eurographics Symposium on Parallel Graphics and Visualization, 2007, 29-36

[6] Wagner T. Corrêa, James T. Klosowski, Cláudio T. Silva. Out-Of-Core Sort-First Parallel Rendering for Cluster-Based Tiled Displays. Parallel Computing, 2003, Volume 29, Issue 3, 325-338

[7] Xavier Cavin, Christophe Mion. Pipelined Sort-last Rendering: Scalability, Performance and Beyond. 6th Eurographics Symposium on Parallel Graphics and Visualization – EGPGV 2006, 2006

[8] M. Makhinya, S. Eilemann, R. Pajarola. Fast Compositing for Cluster-Parallel Rendering. Eurographics Symposium on Parallel Graphics and Visualization, 2010

[9] Steven Molnar, Michael Cox, Devid Ellsworth, Henry Fuchs. A Sorting Classification of Parallel Rendering. IEEE Computer Graphics and Applications, 1994, 14, 23-32

[10] Voicu Popescu, Anselmo Lastra, John Eyles. Sort-First Parallelism for Image-Based Rendering.

[11] Tobias Ritschel, Thorsten Grosch, Hans-Peter Siedel. Approximating Dynamic Global Illumination in Image Space. Proceedings ACM SIGRAPH Symposium on Interactive 3D Graphics and Games in Boston, 2009

[12] Jérémie Allard, Bruno Raffin. A Shader-Based Parallel Rendering Framework. IEEE Visualization 2005 conference proceedings, 2005

[13] Thomas Akenine-Möller, Eric Haines, Naty Hoffman. Real-Time Rendering. CRC Press, 2008

[14] Matt Pharr (Editor). GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation. Addison-Wesley Professional, 2005

[15] Hubert Nguyen (Editor). GPU Gems 3. Addison-Wesley Professional, 2007

[16] NVIDIA Corporation. SLI – Introduction to SLI. NVIDIA GameWorks Documentation, 2015, URL: http://docs.nvidia.com/gameworks/content/technologies/desktop/sli.htm

[17] NVIDIA Corporation. Introducing SLI antialiasing: The Ultimate in Visual Quality. URL: http://www.nvidia.com/object/slizone_sliAA_howto1.html

[18] Thomas W. Crockett. An introduction to parallel Rendering . Parallel Computing, 1997, Volume 23, Issue 7, 819-843

[19] Bengt-Olaf Schneider. Parallel Polygon Rendering. IEEE Visualization 2000, 2000

[20] Ulrich Neumann. Parallel volume-rendering algorithm performance on mesh-connected multi-computers. ACM SIGGRAP Symposium on Parallel Rendering 1993, 1993, 97-104

[21] Frederic I. Parker. Simulation and expected performance analysis of multiple processor Z-buffer systems. Proceeding SIGGRAPH 1980of the 7th annual conference on Computer graphics and interactive techniques, 1980, 48-56

[22] B. Moloney, M. Ament, D. Weiskopf, T. Moller. Scalable Sort-First Parallel Direct Volume Rendering with Dynamic Load Balancing. Eurographics Symposium on Parallel Graphics and Visualization, 2007

[23] Stefan Eilemann, Ahmet Bilgili, Marwan Abdellah, Juan Hernando, Maxim Makhinya, Renato Pajarola. Parallel Rendering on Hybrid Multi-GPU Clusters. Eurographics Symposium on Parallel Graphics and Visualization (2012), 2012

[24] Ivan E. Sutherland, Robert F. Sproull, Robert A. Schumacker. A Characterization of Ten Hidden Surface Algorithms. Computing Surveys 6(1), 1974, 1-55

[25] Parashar Krisnamachari. Global Illumination in a Nutshell. The Polygoners, URL: http://www.thepolygoners.com/tutorials/GIIntro/GIIntro.htm, Stand: 02.04.2016

[26] Igor Wallossek, Greg Ryder. Micro-Stuttering and GPU Scaling in CrossFire and SLI, URL: http://www.tomshardware.com/reviews/radeon-geforce-stutter-crossfire,2995.html, Stand: 02.04.2016

[27] Shawn Hargreaves, Mark Harris. Deferred Shading – What deferred shading is and how to implement it. 6800 Leagues Under the Sea, 2004

[28] Simon Green. NVIDIA SLI Multi-GPU. GameDevelopers Conference 2005 San Francisco, 2005

[29] Tom Loske. 14-Megapixel-Gaming: Multi-GPU-Benchmarks mit GTX 97, GTX980, R9 290X und mehr. PC GAMES HARDWARE, URL: http://www.pcgameshardware.de/Grafikkarten-Grafikkarte-97980/Specials/SLI-Crossfire-GTX-970-GTX-980-R290X-1149715/, Stand: 02.04.2016

[30] OctaneBench Results. URL: https://render.otoy.com/octanebench/results.php, Stand: 02.04.2016

[31] Rudrajit Samanta, Thomas Funkhouser, Kai Li, Jaswinder Pal Singh. Sort-first parallel rendering with a cluster of PCs. SIGGRAPH 2000 Technical sketches, 2000

[32] Rudrajit Samanta, Thomas Funkhouser, Kai Li, Jaswinder Pal Singh. Hybrid sort-first and sort-last parallel rendering with a cluster of PCs.HWWS ’00 Proceedings of the ACM SIGGRAPH/EUROGRAHICS workshop on Graphics hardware, 200, 97-108

[33] Scott Wasson. ATI's CrossFire dual-graphics solution. URL: http://techreport.com/review/8826/ati-crossfire-dual-graphics-solution, Stand: 02.04.2016

[34] Paul Rademacher. Ray Tracing: Graphics of the Masses. URL: https://www.cs.unc.edu/~rademach/xroads-RT/RTarticle.html, Stand: 03.04.2016

Anlagen

- CD mit der beispielhaften Implementierung eines Hybrid-Sort-First-Sort-Last-Verteilungsverfahrens

Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe. Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Ort, Datum Vorname Nachname